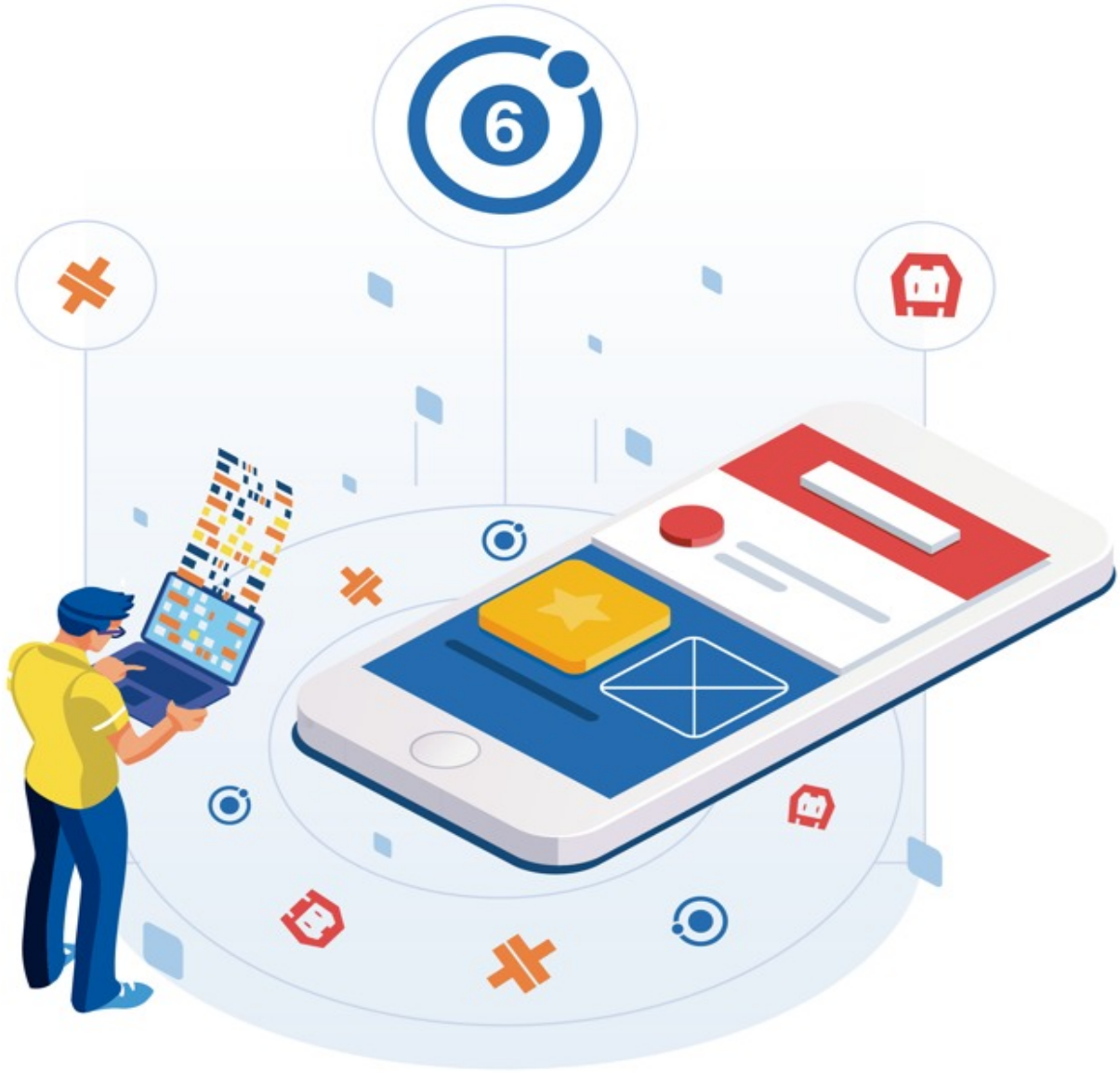
Ionic 6

Desenvolvimento multiplataforma para dispositivos móveis

Edição atualizada



Casa do Código | alura

EVERTON COIMBRA DE ARAÚJO

Sumário

- ISBN
- Agradecimentos
- Sobre o autor
- Prefácio
- Sobre o livro
- 1. Dispositivos móveis, desenvolvimento cross-platform e Ionic
- 2. Ionic — Instalação e testes
- 3. A primeira aplicação Ionic
- 4. Listagem de dados e navegação entre páginas
- 5. Persistindo dados fisicamente em aplicações Ionic
- 6. Sidemenu, CRUD com SQLite e o componente Select
- 7. CRUD com Firestore para acesso a dados remoto e uma página de pesquisa
- 8. Câmera e álbum de fotos no CRUD de clientes e Storage do Firebase
- 9. Autenticação e criação de usuários no Firebase e uso de Tabs
- 10. Integração com Google Maps
- 11. Os estudos não param por aqui

ISBN

Impresso: 978-65-86110-96-8

Digital: 978-65-86110-12-8

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

Este é um espaço que considero muito importante em meus livros. É o momento de reconhecimento ao apoio direto ou indireto de pessoas importantes para mim, durante o processo de pesquisa e escrita do livro.

Não posso jamais deixar de agradecer ao apoio dos meus filhos, pessoas-chave em minha vida. No momento em que me encontro, pessoal e profissionalmente, duas pessoinhas são muito importantes para os momentos de relaxamento: Maria Clara e Vicente Dirceu. Meus filhos, minhas joias mais preciosas. Meu filho Gabriel, conversamos quase todos os dias e, na maioria das vezes, é você quem me dá conselhos. Ainda tenho o Yuri e a Yara. A Yara me presenteou com uma joia maravilhosa, minha neta, Amelie, linda e maravilhosa.

Uma vez mais meu sincero e grande agradecimento à excelente pessoa, competentíssimo profissional: Leonan Fraga Leonardo, que mantém sempre seu LinkedIn atualizado. O Leonan tem sido um grande amigo, parceiro de testes, que em muitos momentos tira minhas dúvidas sobre problemas que ele já resolveu. Hoje o cara está muito bem colocado em uma grande multinacional de TI. Focado em aplicações web e dispositivos móveis, ele está sempre disponível para trocar informações.

Embora este texto seja repetitivo, sua força só aumenta cada vez que o escrevo. Agradeço muito à Casa do Código, à Vivian Matsui e Sabrina Barbosa, dois anjos que sempre me acompanham nos trabalhos que desenvolvo, sempre pacientes e procurando tempo para ler e contribuir com as revisões dos livros. Estes agradecimentos se estendem a toda a equipe da editora.

De maneira geral, agradeço sempre aos meus colegas de trabalho do Departamento de Computação da UTFPR, campus Medianeira, meu paraíso profissional, pelo coleguismo e pelos ouvidos.

Não poderia concluir os agradecimentos sem fazê-los também aos meus alunos e alunas. Queridas pessoinhas que estão aos poucos se soltando mais e vendo que a cara feia não é de bravo, mas sim de alguém que deseja que tenham um futuro bom, respeitoso e com muito orgulho de ser programador ou programadora (ou *DEV*).

Sobre o autor

Everton Coimbra de Araújo atua na área de ensino de linguagens de programação e desenvolvimento. É tecnólogo em processamento de dados pelo Centro de Ensino Superior de Foz do Iguaçu, possui mestrado em Ciência da Computação pela UFSC e doutorado pela UNIOESTE em Engenharia Agrícola, na área de Estatística Espacial. É professor da Universidade Tecnológica Federal do Paraná (UTFPR), campus Medianeira, onde leciona disciplinas no curso de Ciência da Computação e de mestrados. Já ministrou aulas de Algoritmos, Técnicas de Programação, Estrutura de Dados, Linguagens de Programação, Orientação a Objetos, Análise de Sistemas, UML, Java para Web, Java EE, Banco de Dados e .NET. Tem se dedicado de corpo e alma ao desenvolvimento mobile.

Acesse https://linktr.ee/evertoncoimbradearaujo e saiba mais sobre o trabalho do autor.

Prefácio

O professor Everton Coimbra de Araújo já escreveu diversos livros na área de informática, que são utilizados em diversas universidades como referência nas disciplinas. Eu mesmo já utilizei alguns como guia para ministrar minhas aulas na UTFPR campus Medianeira, e sempre tive bons resultados.

O diferencial dos livros do Everton é que ele tem conhecimento didático sobre os assuntos que escreve e sempre faz referência ao dia a dia nas empresas, o que torna a obra muito mais interessante.

O assunto abordado neste livro é realmente o que profissionais da área de desenvolvimento estavam precisando para desenvolver seus aplicativos para dispositivos móveis.

Há pouco tempo, quando se decidia desenvolver aplicativos, deveria se optar para o sistema operacional Android ou iOS, sendo despendido muito esforço para adaptar para os dois sistemas operacionais. Com o framework apresentado neste livro, o Ionic, o desenvolvedor deverá despender esforços apenas na sua aplicação, sendo transparente a questão do sistema operacional.

Lendo este livro me resta pouca saudade do desenvolvimento de aplicativos para Palmtop, PDAs ou HandHelds. Como ficou fácil desenvolver aplicativos para os usuários trabalharem remotamente sem necessidade de conexão através dos modens ou levar o equipamento até a empresa para transferência dos dados.

O autor foi muito feliz na forma como escreveu este livro. Nos capítulos iniciais, faz uma contextualização de dispositivos móveis e ferramentas utilizadas, de forma que o leitor não precisa ser um expert. Em seguida, é apresentada a instalação e teste da plataforma que será utilizada. Ele vai explicando o assunto e ligando a leitura aos conteúdos dos capítulos seguintes de uma forma fácil, didática e interessante de aprendizado.

No final da leitura deste livro você estará apto para desenvolver bons aplicativos comerciais, utilizando uma das melhores ferramentas disponíveis no mercado para dispositivos móveis. Uma boa leitura.

César Angonese

Universidade Tecnológica Federal do Paraná - Campus Medianeira

Sobre o livro

Este livro traz, na prática, o desenvolvimento de aplicações *cross-platform* com o Ionic, um framework para desenvolvimento de aplicativos para dispositivos móveis e web. Desenvolver um aplicativo para ser publicado em dispositivos com plataformas diferentes (iOS e Android) é uma tarefa tranquila com o Ionic. É possível criar uma aplicação utilizando a linguagem HTML e TypeScript, e ela ser publicada para as duas plataformas, além do navegador.

O livro é desenvolvido em dez capítulos. O primeiro é apenas teórico, mas não menos importante, pois trago nele contextualizações sobre dispositivos móveis e as ferramentas usadas no livro. Já no segundo, apresento o processo de instalação e teste da plataforma e do IDE que utilizaremos, o Visual Studio Code. Embora o Ionic traga recursos para publicação de aplicações em plataformas diferentes que o iOS e Android, o foco no livro será para estas duas.

A aplicação a ser implementada durante o livro refere-se a uma oficina mecânica, com cadastro de clientes, serviços, peças e atendimentos. Persistiremos os dados localmente, no navegador, em uma execução da aplicação em ambiente web. Depois em uma base de dados local e, ao final, faremos uso de recursos do Firebase. Ofereceremos ao usuário a possibilidade de capturar fotos no momento do cadastro do cliente.

A prática começa bem legal no capítulo 3, onde criaremos interfaces com o usuário para interação com a aplicação. Veremos alguns recursos possíveis para tratamento dos dados informados, validando e informando ao usuário possíveis inconsistências. Veremos nossos primeiros controles visuais e buscaremos entender a estrutura de aplicações Ionic.

No quarto capítulo, trabalharemos a navegação entre páginas, criaremos uma página para listagem de dados e a partir desta

página possibilitaremos a navegação para outra. Também teremos a introdução em serviços na concepção de componentes do Ionic. Teremos neste capítulo um CRUD completo, onde poderemos inserir, recuperar, atualizar ou remover dados.

No capítulo 5, trabalharemos a persistência física de dados, iniciando com o Storage do Ionic, que mantém os dados no navegador, onde teremos nossos primeiros exemplos. Após esse conhecimento, faremos uso de um banco de dados, o SQLite, também para a persistência física dos dados. Para os dois cenários teremos as operações CRUD implementadas.

No capítulo 6, traremos o conceito de Sidemenu do Ionic, que é uma página que serve como um menu de opções disponibilizadas pela aplicação para o usuário. Começaremos aqui o trabalho de associação de classes de negócio em nossas páginas de interação com o usuário.

Dando sequência ao tema de persistência, no capítulo 7, conheceremos o Firebase e vamos utilizar o Firestore, que é uma solução do Google para persistência e sincronização de dados em aplicações web e para dispositivos móveis. Um capítulo muito legal.

No capítulo 8, vamos interagir com a câmera e álbum de fotos de nossos dispositivos físicos. Possibilitaremos ao usuário capturar uma foto de um cliente ou selecioná-la de seu álbum de fotografias. Inicialmente gravaremos esta imagem em nosso dispositivo. Depois, no mesmo capítulo, faremos uso de outra ferramenta da Google, o Storage, que é capaz de armazenar imagens na nuvem. Veremos as operações CRUD também para essas imagens e as associaremos com nossos clientes.

Já nos aproximando do final do livro, veremos uma nova maneira de navegação entre páginas oferecidas pelo Ionic, as guias (Tabs). Para ilustrar seu uso, trabalharemos com autenticação do usuário e, uma vez mais, recorrendo ao Firebase, utilizaremos a ferramenta Authentication e você verá que é muito simples e flexível.

Enfim, como último capítulo, o de número 10 traz a integração com o Google Maps. Pesquisaremos endereços e buscaremos nossa localização. Fechando o capítulo e o livro, veremos como alterar a imagem de inicialização de nossa aplicação e o ícone que ela terá em nosso dispositivo.

Certamente, este livro pode ser usado como ferramenta em disciplinas que trabalham o desenvolvimento de dispositivos móveis, quer seja por acadêmicos ou professores. Isso porque ele é o resultado da experiência que tenho em ministrar aulas dessa disciplina, então trago para cá os anseios e as dúvidas dos alunos que estudam comigo. Eu já utilizo este conteúdo com meus alunos e eles gostam muito.

É importante que você tenha conhecimento de Orientação a Objetos e de alguma linguagem de programação, mas não é um fator impeditivo. Conhecimentos básicos sobre banco de dados também são interessantes. O repositório com todos os códigos-fontes usados no livro pode ser encontrado em: https://github.com/evertonfoz/implementacoes-de-livros/tree/master/ionic.

Os arquivos disponibilizados no GitHub estão de acordo com as versões apontadas no livro. Recomendo que, inicialmente, você implemente os exemplos nestas versões e, após o sucesso, tente atualizar para a mais recente, já que é possível que haja essa atualização quando você estiver lendo este livro, uma vez que a equipe do Ionic é muito dinâmica e sempre está disponibilizando atualizações evolutivas e corretivas. Reforçando, fique atento ao caso de atualizar os plugins e componentes dos projetos, pois essa tecnologia é dinâmica e atualizações estão sempre ocorrendo.

Que a leitura deste livro seja para você tão prazerosa como para mim foi escrevê-lo. Desfrute, sem moderação. Sucesso.

CAPÍTULO 1
Dispositivos móveis, desenvolvimento cross-platform e Ionic

Olá! Seja bem-vindo ao primeiro capítulo deste livro. Ele será curto e conterá apenas teoria, mas nos demais compensarei com a prática. Nele, buscarei trazer conteúdo sobre o panorama dos dispositivos móveis, tipos de dispositivos, suas características e opções para o desenvolvimento de aplicativos.

As ferramentas que serão usadas neste livro serão apresentadas conforme forem necessárias. É importante que você tenha conhecimento do paradigma de Orientação a Objetos e de alguma linguagem de programação Orientada a Objetos para um perfeito acompanhamento do desenvolvimento proposto para este livro. O desejável é um conhecimento ainda que básico em tecnologias WEB, como HTML, CSS e JavaScript, entretanto este não é um requisito proibitivo para você ler este livro.

A aplicação proposta para ser desenvolvida neste livro, a partir do terceiro capítulo, refere-se ao atendimento oferecido por uma oficina mecânica. A aplicação subsidiará o cadastro de clientes, peças e serviços a serem contratados para veículos e registro de atendimento a clientes. Nessas funcionalidades, serão trabalhados tipos de telas para a aplicação, controles de entrada de dados, listagem de dados, templates para aparência da aplicação, acesso a banco de dados, consumo de serviços web e uso de câmera.

Não faz parte do escopo do livro ter essa aplicação implementada em sua totalidade, mas sim para a aplicação do Ionic em algumas funcionalidades relacionadas a este problema proposto.

Durante o desenvolvimento da aplicação, diversas técnicas e recursos serão utilizados para subsidiar uma boa arquitetura para o desenvolvimento da aplicação.

1.1 Os dispositivos móveis na atualidade

Antes de começarmos esta seção, permita-me um alerta. A contextualização proposta aqui é semelhante à que utilizei em demais livros sobre dispositivos móveis aqui da Casa do Código. Optei por isso por estar trabalhando uma mesma situação. Dessa maneira, nesta e na próxima seção, você pode ter um sentimento de *déjà-vu*, mas é só nelas, ok? Depois, tudo é com base no Ionic.

Até anos atrás, era comum conhecer pessoas que adquiriam computadores apenas para navegar na internet, ler e-mails e acessar algumas aplicações para leitura de livros, artigos ou documentos diversos. No lado corporativo empresarial, notebooks eram fornecidos aos colaboradores para que desempenhassem algumas atividades, como registro de uma venda para um cliente, recebimento de uma conta, anotação de um pedido e agendamento de compromissos, dentre diversas outras atividades.

Com o surgimento dos dispositivos móveis, a venda de computadores pessoais tem sofrido constantes quedas. Isso começou lá atrás, de maneira modesta e quase despercebida, com handhelds, palmtops e PDAs (*Personal Digital Assistants*), mas que começou a ganhar destaque com a chegada do iPod e, depois, dos iPhones. A massificação veio por meio do Google, com o desenvolvimento do sistema operacional Android para dispositivos mais **ace$$ívei$**, em relação aos produtos da Apple.

Embora um dispositivo móvel permita atendermos a necessidades pessoais e corporativas, é importante ressaltar que a maneira como este mercado se desenvolveu foi bem distinta. Tem-se o lado recreativo, em que é possível ter em seu dispositivo diversos jogos; o cultural, que permite o acesso a filmes e livros; o da organização pessoal, com diversos recursos para gestão financeira e de compromissos, acesso a bancos, compras; e a parte empresarial, com aplicativos corporativos, de gestão, operacional das empresas e vendas. Há ainda pessoas que utilizam esse dispositivo apenas para

navegar na internet e acessar redes sociais, dispensando, em muitos casos, a necessidade de um computador.

Toda essa popularização e mudança comportamental resultaram no que é visto atualmente como um fenômeno BYOD (*Bring Your Own Device* — Traga seu Próprio Dispositivo). Ou seja, o colaborador está levando para seu ambiente de trabalho um dispositivo potente (o seu) e que pode, em alguns casos, ser substituído por um computador.

Com isso e com o acesso à rede corporativa liberado, todos podem ter no bolso as aplicações de seu local de trabalho. Isso pode gerar mais produtividade também (ou não **:-)**). Porém, no que diz respeito à segurança, traz novas ameaças e vulnerabilidades, mas isso é assunto para a equipe de infraestrutura.

Pense em trabalhos que exigem atividade externa, quer seja em campo ou locais urbanos, como a função de um engenheiro agrônomo ou um vendedor externo. Esses profissionais precisavam se dirigir à empresa, buscar blocos ou formulários impressos específicos para coleta de dados, retornar aos seus escritórios e registrar os dados coletados. Com um dispositivo móvel, ele não precisa nem ir ao seu escritório, pois seu equipamento já permite o registro de sua coleta ou venda no momento em que ela ocorre. Você também já reparou onde os atendentes de restaurantes registram nossos pedidos? Em um dispositivo móvel.

E se o acesso à internet não for possível no momento da atividade, é possível uma sincronização com os servidores da empresa tão logo uma conexão seja possível. Os dispositivos também podem estar conectados apenas a uma rede interna, não sendo necessária a internet.

As empresas buscam também tirar proveito dessa realidade, quer seja dando maior liberdade para seus colaboradores, ou economizando em despesas, como energia, água, condomínio ou aluguel, dentre outras.

Neste livro, trabalharemos aplicações que possam ser executadas, com recursos nativos, nos sistemas operacionais iOS e Android. Neste último, existem diversos dispositivos, nas mais diversas especificações de tamanho e resolução de tela, recursos e processamento.

1.2 O desenvolvimento móvel cross-platform

Não é porque os dispositivos móveis estão em uma grande ascendência que os computadores pessoais se tornaram descartáveis. Existem ainda diversas atividades e aplicativos que precisam ser realizados e utilizados em um computador que possua um teclado (que não seja na tela), uma tela grande (às vezes mais do que uma) e um desempenho ainda não alcançado pelos dispositivos móveis atuais.

Com isso posto, é importante estar ciente de que as aplicações comerciais trazidas para o ambiente móvel precisam, nas interações com o usuário, ter ou solicitar dados e informações que possam caber "confortavelmente" na tela que será usada. Nada de encher de informações desnecessárias para o processo atual.

O desenvolvimento para dispositivos móveis, até a presente data, vem sendo focado nas plataformas iOS e Android, que são as mais utilizadas. Cada uma dessas plataformas possui suas próprias características e recursos. Com isso, você pensa em escolher uma plataforma para que seu aplicativo funcione e limitar o uso a apenas esta?

Se for uma empresa que especifique a plataforma e se o aplicativo deve funcionar apenas entre os colaboradores dela, isso é uma opção sim. Mas e se sua aplicação precisar ser utilizada em plataformas diferentes? Aí esta opção já não poderia ser escolhida.

O que vem ocorrendo no desenvolvimento para dispositivos móveis é que, ao desenvolver uma aplicação para um dispositivo e uma plataforma, esperamos que esta aplicação possa funcionar em qualquer dispositivo, de qualquer plataforma.

Por exemplo, deseja-se que uma aplicação desenvolvida para o iPhone 12 Pro Max possa funcionar da mesma maneira em um smartphone com Android. O que fazer para resolver este problema? Montar várias equipes de desenvolvimento, cada uma usando um ambiente de desenvolvimento diferente e focada em uma plataforma? Desenvolver diversas versões do mesmo aplicativo? Isso não é produtivo e nem barato, concorda?

O objetivo é partir para uma ferramenta que possibilite o desenvolvimento *cross-platform*. Existem várias. Algumas geram o aplicativo para ser executado em um ambiente específico, como se fosse uma máquina virtual (conhecido como ambiente híbrido). Outras ferramentas geram código para ser executado em um ambiente web, como se fosse um site em um navegador (que é o caso do Ionic) e, temos ainda, ferramentas que geram aplicativos nativos para as plataformas escolhidas, como o Xamarin e o Flutter. O Ionic, por meio de frameworks específicos, como *Cordova* e *Capacitor*, possibilitam a *transformação* de uma aplicação Ionic em uma aplicação híbrida. Veremos isso a partir do próximo capítulo.

O site do Ionic oferece um livro gratuito chamado *Hybrid Vs. Native*, que, durante a escrita deste livro, está sendo disponibilizado na página principal do Ionic, a https://ionicframework.com/. Quando tiver tempo, talvez seja interessante lê-lo. É bem pequeno.

Um ponto importante é verificar na App Store (iOS) e no Google Play os padrões e as regras que devem ser respeitados para a publicação oficial de sua aplicação. Por exemplo, os ícones das aplicações são diferentes em cada plataforma, assim como a barra de navegação e o padrão de cores. São alguns dos pontos que você deve verificar e garantir que sua aplicação os respeite para que a publicação seja

aceita. E para você distribuir aplicações pela App Store e Play Store, será necessário ter uma conta de desenvolvedor, que é paga.

1.3 O Ionic Framework

De acordo com o próprio portal do Ionic, ele é definido como um framework, de código aberto, de ferramentas para serem utilizadas para o desenvolvimento da interface com o usuário, que permite um bom desenvolvimento de aplicações baseadas em tecnologias para web, como JavaScript, HTML e CSS. Outra definição, do próprio portal, traz o Ionic como uma biblioteca de componentes de interface com o usuário (UI).

Embora o Ionic seja *open source* e permita o desenvolvimento de aplicações com ferramentas gratuitas, ele possui pacotes com recursos adicionais pagos - e que talvez você julgue interessante. Quando puder, verifique em https://ionic.io/pricing.

Como dito no primeiro parágrafo desta seção, o Ionic tem como foco o desenvolvimento de aplicações baseadas em tecnologia para web, que executam em um navegador, ou ainda em dispositivos de plataforma específica. Desta maneira, o desenvolvimento cross-platform, por meio do Ionic, faz uso do modelo híbrido de aplicações móveis. Ou seja, ele permite que aplicações desenvolvidas em Ionic sejam executadas em dispositivos móveis, mas não fazendo uso direto de recursos nativos, embora, como veremos, isso seja possível por meio de componentes de terceiros. As aplicações Ionic são conhecidas como *Webview*, ou ainda, *Frontend Application*.

Como uma plataforma para desenvolvimento de aplicativos cross-platform, o Ionic tem como foco dispositivos móveis com o iOS e Android e, quando executarmos nossa primeira aplicação, os veremos em seus respectivos pseudoemuladores, gerados pelas ferramentas do Ionic.

A linguagem de programação que adotaremos, e a padrão para o desenvolvimento em Ionic, é a TypeScript, que possibilita um desenvolvimento estático, com compilação em tempo de implementação, garantindo que o código JavaScript que será gerado esteja livre de erros sintáticos. Também utilizaremos o HTML e CSS em nossas camadas de apresentação, e você verá que estes recursos abrem um enorme leque de customização da interface com o usuário. Estes conhecimentos não são excludentes para quem deseja ler este livro, pois fornecerei explicações que sejam suficientes para o acompanhamento das atividades, buscando sempre deixar referências que possam complementar seu conhecimento.

O TypeScript, de acordo com seu próprio portal, é um superconjunto tipado que é compilado para JavaScript puro. Todo código TypeScript que implementarmos, após uma compilação bem-sucedida, será gerado para código JavaScript, que é o código que será utilizado pela aplicação no dispositivo. Embora a linguagem foco seja TypeScript, teremos implementações em JavaScript puro também, o que é perfeitamente válido para código TypeScript.

O TypeScript foi desenvolvido como uma iniciativa da Microsoft, por Anders Hejlsberg, criador do Turbo Pascal, Delphi e C#. Vem sendo uma linguagem/ferramenta adotada de forma cada vez maior pela comunidade de desenvolvedores JavaScript. Ela permite inclusive o desenvolvimento Orientado a Objetos, fazendo uso de classes, interfaces, herança e métodos, dentre outras características deste paradigma. Conforme formos utilizando esses recursos durante nossas aplicações, procurarei contextualizá-los.

De acordo com relatos identificados na web, o Ionic sofreu mudanças drásticas na passagem da versão 1 para a 2 e, consequentemente, para a 3 e 4. Da 4 para a 5 e 5 para 6, algumas mudanças também ocorreram, mas acredito que seja para uma maturidade necessária. Essas atitudes causou e ainda causam uma série de desconfortos na comunidade desenvolvedora, pois gerou e gera muita mão de obra para atualização de suas aplicações,

havendo casos até de pessoas que optaram por mudar de plataforma. A versão 4 começou o processo de trazer para o Ionic arquiteturas já validadas pela comunidade, como Angular, Vue e React, já disponíveis a partir da versão 5. Nós utilizaremos o Angular neste livro. No próximo capítulo, comentarei ele de maneira mais descritiva.

Conclusão

Conforme prometido no início deste capítulo, ele foi breve. Eu poderia trazer muito mais teoria aqui, mas penso que você quer começar logo com a prática. Entretanto, precisava lhe apresentar o texto aqui trabalhado.

É importante ler um pouco sobre dispositivos móveis, cross-platform, TypeScript, JavaScript, HTML e CSS. É claro que o que eu trouxe para você foi apenas uma introdução, mas acredite, com ela você já está pronto (a) para começar o desenvolvimento móvel e você tem motivos para escolher o Ionic. Isso porque você quer e precisa desenvolver para, no mínimo, duas plataformas e posso assumir que não quer, ou não pode, ter uma equipe para cada plataforma.

Se você precisa de produtividade e quer que sua aplicação seja desenvolvida em uma única plataforma, mas que seja executada nas duas mais usadas atualmente, então a solução que proponho neste livro é o Ionic.

No próximo capítulo, instalaremos a ferramenta e testaremos seu funcionamento. Até lá.

Capítulo 2
Ionic — Instalação e testes

O processo de preparação do ambiente de trabalho para desenvolvimento e distribuição de aplicações Ionic em dispositivos móveis exige a instalação de algumas ferramentas que são pré-requisitos. Neste segundo capítulo, veremos essas ferramentas e algumas possibilidades para execução e testes de aplicações. Também instalaremos o ambiente para a codificação de nossas práticas, o IDE, que será o Visual Studio Code.

Com as ferramentas e dependências instaladas, trabalharemos a criação e execução de uma aplicação simples para validar a corretude desse processo. Trabalharemos a distribuição da aplicação em dispositivos físicos e emuladores, por meio de frameworks que realizam esta ligação de código front-end com os SDKs nativos de cada plataforma. Veremos o uso do Capacitor, embora ainda seja possível fazer uso do Cordova. Temos aqui um capítulo relativamente simples, mas que envolve um certo esforço e atenção em todo o processo.

2.1 Ferramentas, downloads e instalação

Estou usando dois equipamentos para desenvolvimento. Um i7, com 32GB de RAM e Windows 10 com todos os updates realizados e um MacBook Air, M1, com 8GB de RAM, executando o macOS Monterey, devidamente atualizado. Todas as ferramentas utilizadas nestes equipamentos podem também ser instaladas em plataforma Linux,

seguindo sempre as orientações específicas para ela, que são sempre fornecidas nos portais dessas ferramentas.

Como dito no início do capítulo, para instalarmos e termos o Ionic disponível para nosso desenvolvimento, precisamos antes realizar o download de algumas dependências. A primeira delas será o `Git`, que é um sistema de controle de versões distribuído. O Git é muito usado entre desenvolvedores, podendo versionar qualquer tipo de documento. Em conjunto com o Git, é muito comum a associação com o GitHub, portal que hospeda um imenso número de projetos, componentes, aplicações e documentos.

É possível obter o Git para download em https://git-scm.com/downloads/ e, no momento da escrita deste livro, a versão disponível é a 2.34.1. Realize o download e a instalação deste nosso primeiro pré-requisito. Caso não queira se preocupar, basta seguir as sugestões do padrão de instalação. Se quiser se aprofundar em cada opção, recomendo uma leitura no portal da ferramenta. Caso não conheça o Git e o GitHub, recomendo os vídeos do meu canal disponíveis em https://www.youtube.com/watch?v=-KcIGuwkoaA&t=3s.

Necessitamos também do Java (JDK) instalado em nossa máquina, pois, no desenvolvimento para Android, o Java é um pré-requisito. A partir da versão 5 do Ionic, a utilização do `Capacitor` é a recomendada, ao contrário das anteriores, que priorizavam o uso do Cordova. A versão do Capacitor que utilizaremos é a 3.3.3. Recomendo que você possa fazer um tour em https://capacitorjs.com/, para ver os processos de instalação, migração do Cordova, plugins disponíveis, mas isso pode ser feito no seu tempo.

Em relação ao Java, é preciso a versão 1.8.X do JDK, que você pode baixar em https://www.oracle.com/technetwork/pt/java/javase/downloads/jdk8-downloads-2133151.html. Será preciso que você coloque o caminho no PATH de sua máquina, incluindo a pasta `bin`, e você pode ver

orientações para isso aqui: https://www.java.com/pt_BR/download/help/path.xml. Embora o Java esteja em versões mais atualizadas, reforçando, é a 1.8.X que deve estar em sua máquina.

Para trabalharmos com a plataforma Android, por meio de dispositivos ou emuladores, eu particularmente recomendo a instalação do Android Studio, pois ele faz todas as configurações necessárias para o nosso desenvolvimento. É possível instalar apenas os recursos necessários caso você queira, mas por complexidades que tive em relação a isso, buscando uma solução mais simples e efetiva, adotei a instalação desta ferramenta, que pode ser obtida em https://developer.android.com/studio. No momento da escrita deste livro, ela está na versão Artic Fox | 2020.3.1.

Com o término da instalação, execute o `Android Studio` e na janela que se abre, selecione o botão `More Actions` e nele a opção `SDK Manager`, tal qual mostra a figura a seguir.

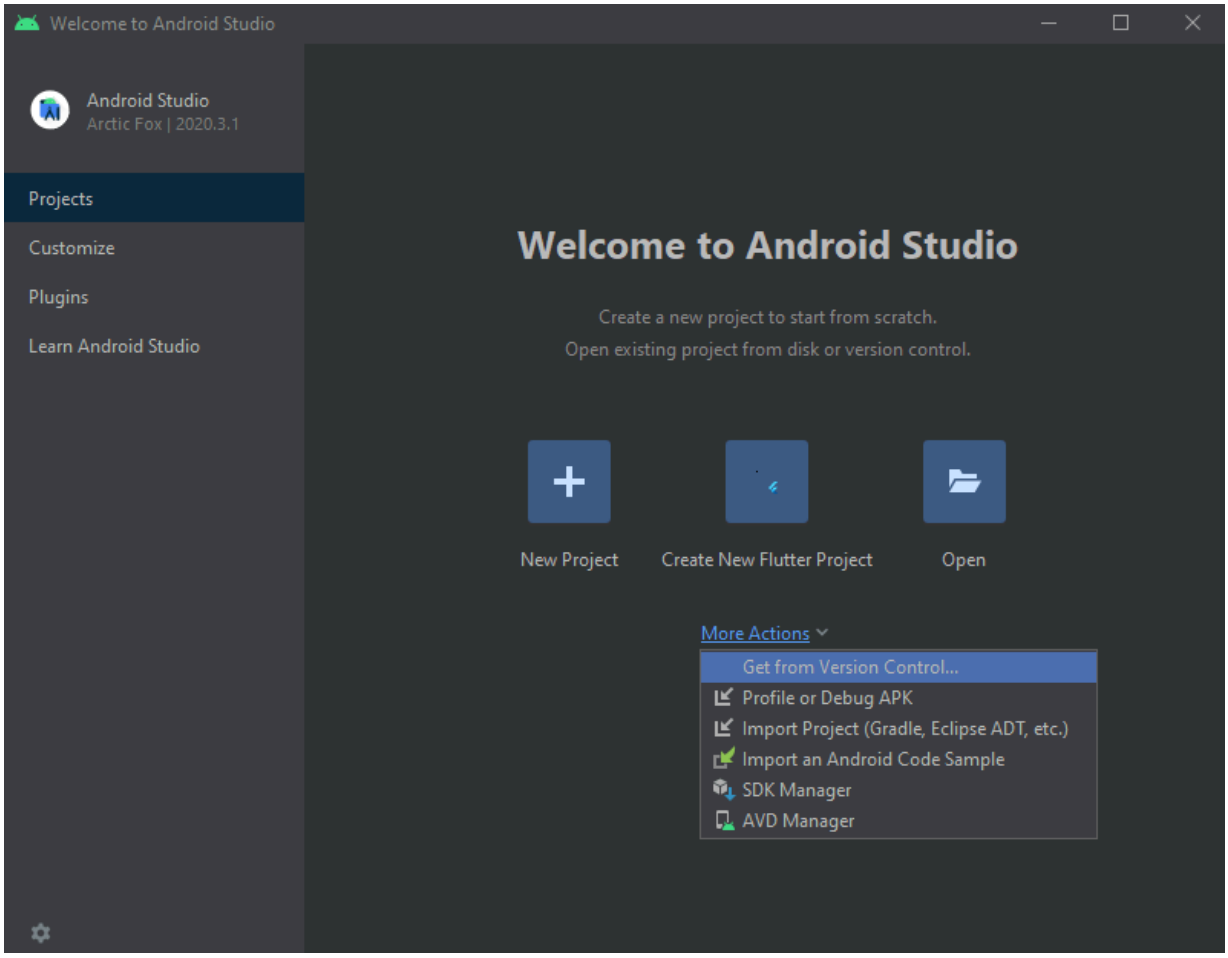


Figura 2.1: Acessando o gerenciador do Android SDK.

Na nova janela que se abre, serão exibidas as plataformas disponíveis para instalação. Eu recomendo que você instale pelo menos as três últimas versões, em meu caso foram as APIs 31, 30 e 29.

Após a seleção, confirme a instalação clicando no botão `Apply`. Esse processo pode demorar um pouco, pois ele depende de conexão com a internet. Observe e guarde o endereço que aparece no topo desta janela, que é onde o SDK foi instalado, porque precisaremos dele.

Precisamos agora baixar e instalar em nossa máquina o(s) emulador(res) onde executaremos nossa aplicação Android, mas, sempre que puder, faça uso de dispositivos físicos para o

desenvolvimento, é bem mais rápido. Como baixamos o Android Studio, vamos fazer uso dele para este processo de instalação de emuladores. Da mesma maneira que acessamos o `SDK Manager` anteriormente, vamos acessar o `AVD Manager` para essa atividade. Ao acessar essa ferramenta você terá acesso a uma janela como a apresentada na figura a seguir.

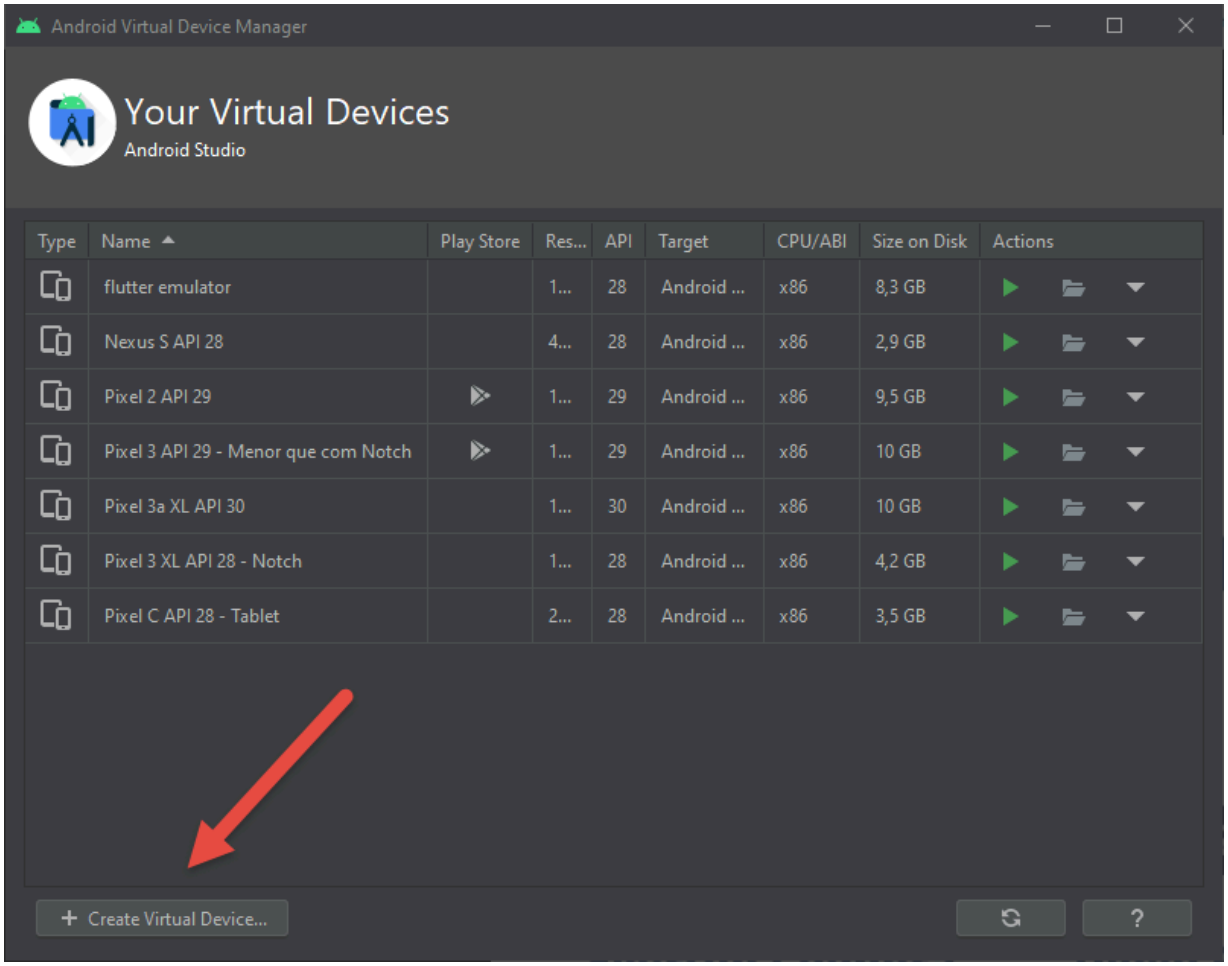


Figura 2.2: Acessando o gerenciador de dispositivos virtuais do Android.

Nesta janela, existe uma relação de dispositivos previamente configurados, que podem ser baixados para nossas máquinas. Pode ser que nenhum dispositivo seja apresentado em seu AVD Manager, o que nos remete a criá-lo. Para isso, clique no botão `Create Virtual Device`. Na janela que se abre, precisamos escolher o modelo do dispositivo que utilizaremos. Eu selecionei o `Nexus 6P` dentro da

categoria `Phone` e então cliquei no botão `Next`. A nova janela traz três guias. Na primeira, precisamos instalar a API para a imagem do dispositivo que estamos criando. Eu selecionei a 30 neste exemplo e realizei o download dela. Ao término do download, precisamos ir para a segunda guia, responsável pela imagem em si do dispositivo. Optei pela x86_64 da API 30 e fiz o download dela. Após a conclusão, retornei para a primeira guia e cliquei no botão `Next`, o que levou para a última janela, para definição do nome do emulador. Informe o que desejar, confirme a operação e você retornará para o AVD Manager, podendo executar o emulador.

Com a configuração dos emuladores concluída, precisamos criar uma variável de ambiente, chamada `ANDROID_SDK_ROOT`, apontando para o SDK do Android, que em minha máquina é `C:\Users\Everton\AppData\Local\Android\Sdk`. Este é o endereço que comentei há pouco para você guardar. Uma vez mais, é possível verificar como criar uma variável de ambiente em https://www.java.com/pt_BR/download/help/path.xml.

Começaremos agora a instalação de ferramentas que usaremos diretamente em nosso desenvolvimento. Começaremos com o `NodeJS`. De maneira simples e rápida, podemos dizer que o `Node.js` é um aplicativo interpretador de código JavaScript, executando no lado servidor da aplicação e não (apenas) no lado cliente, como é comumente utilizado. Segundo o próprio portal do Node.js (https://nodejs.org), a ferramenta tem também o objetivo de auxiliar programadores na criação de aplicações escalonáveis, com enorme capacidade de processamento. Durante a leitura deste livro, faremos uso do `Node` nos momentos em que instalaremos plugins e componentes que nos auxiliem no desenvolvimento, além do próprio Ionic. Também teremos o Node em execução quando formos testar as aplicações que desenvolveremos em Ionic. A versão disponível no momento da escrita deste livro é a 17.2.0, porém, as versões ímpares não são recomendas para produção e quando usei esta versão realmente tive problemas, que exigiram muito esforço para tudo funcionar. Desta maneira, recomendo o uso da versão 16.13.1

LTS. A dica é usar sempre uma versão PAR e que tenha o LTS em seu nome.

Você pode verificar se já tem o `Node` instalado em seu ambiente, bastando, no console (`Command` , `Windows PowerShell` ou `Terminal` , acessando sempre como Adminstrador) digitar `node -v` e `npm -v` . Caso as ferramentas já estejam disponíveis em sua máquina, verifique as informações retornadas e veja se seu ambiente está atualizado. Caso precise atualizar, recomendo baixar novamente o instalador e executá-lo, é mais simples. Entretanto, existe na documentação processos apenas para a atualização, mas deixarei isso para você.

Em relação ao NPM (Node Package Manager - Gerenciador de Pacotes do Node), ferramenta que utilizaremos muito, o portal NodeBR (http://nodebr.com/) traz a informação de que ele pode ser visto de duas maneiras: segundo o portal, a primeira e a mais importante é que o NPM é um repositório on-line para a publicação de projetos de código aberto para o Node.js; em sua segunda visão, mais técnica, é um aplicativo de linha de comando que interage com esse repositório on-line, auxiliando na instalação de pacotes, gerenciamento de versão e gerenciamento de dependências em nossas aplicações. Quando instalamos o Node.js, o NPM é também instalado por padrão.

Retornando ao Ionic, podemos afirmar que as aplicações desenvolvidas por ele são criadas e assistidas por um utilitário de linha de comando, conhecido como `Ionic CLI` , ou Ionic Command Line Interface. O Ionic CLI é a ferramenta que o framework oferece para execução de diversas atividades durante o processo de desenvolvimento, como a criação do projeto da aplicação, criação automatizada de artefatos para seus projetos e execução das aplicações, dentre outras funcionalidades que veremos no desenvolver do livro.

Para instalarmos o Ionic e termos à nossa disposição o Ionic CLI, precisaremos fazê-lo por meio do NPM, que já instalamos em passos

anteriores, junto com o Node.js. Para iniciarmos, é preciso acessar a console do Windows (Command) ou o `Windows PowerShell`, com direitos de administrador. Já no Mac é preciso acessar como `sudo`. Veja o conjunto de instruções a seguir, cada uma precedido de comentários.

```
// Instalação da versão mais atualizada do Ionic/CLI de maneira global no sistema (-g)
npm install -g @ionic/cli

// ou

// Instalação de uma versão específica para o Ionic/CLI, no caso, a 6.18.1, que será a que utilizaremos em todo o livro
npm install -g @ionic/cli@6.18.1
```

Fique atento ao retorno deste processo. É comum haver avisos para a atualização do NPM, mesmo você tendo instalado ele neste momento.

A versão do Ionic instalada em minha máquina para a implementação dos exemplos deste livro foi a mais atualizada no momento da escrita, a 6.18.0. Você pode verificar a versão do Ionic CLI em sua máquina executando a instrução `ionic info` no terminal. Recomendo que mantenha essa versão durante todo o livro. Caso opte por versões diferentes, mais recentes ou anteriores, podem surgir problemas relativos a especificidades de cada uma.

Com o Ionic e suas dependências devidamente instaladas, precisamos agora do ambiente que utilizaremos para o desenvolvimento de nossas aplicações e adotaremos neste livro o `Visual Studio Code`, que tem sido adotado amplamente por comunidades de desenvolvedores de diversas linguagens. Ele é gratuito e realmente muito bom. Você pode realizar seu download pelo link https://code.visualstudio.com/. No momento da escrita deste livro, a versão é a 1.63.0.

Após a instalação e execução do aplicativo, é possível verificarmos atualizações do ambiente pelo menu `Help -> Check for updates`. Em caso de existência de atualizações necessárias, aparecerá o item de menu `Install updates`. Basta seguir as orientações para que seu ambiente seja atualizado.

Como dito no capítulo anterior, utilizaremos o TypeScript como linguagem, pois é a linguagem adotada pelo Ionic. Também foi comentado que o código TypeScript é convertido (automaticamente) para JavaScript (processo de `Transpiling`), que é a linguagem que realmente terá o código executado e, certamente utilizaremos os recursos dela, em conjunto com o TypeScript. Desta maneira, é preciso que nosso ambiente (IDE) seja configurado para essas linguagens, o que nos trará benefícios, como *autocomplete* e *intellisense*. Sendo assim, na primeira execução do ambiente, aparecerá na janela principal a sugestão de instalação de recursos para essas linguagens. Clique nos links oferecidos para que isso seja realizado. Veja a figura a seguir. A guia apresentada pode ser acessada por meio do menu `Help -> Welcome`. Já já começaremos a utilizar este ambiente.

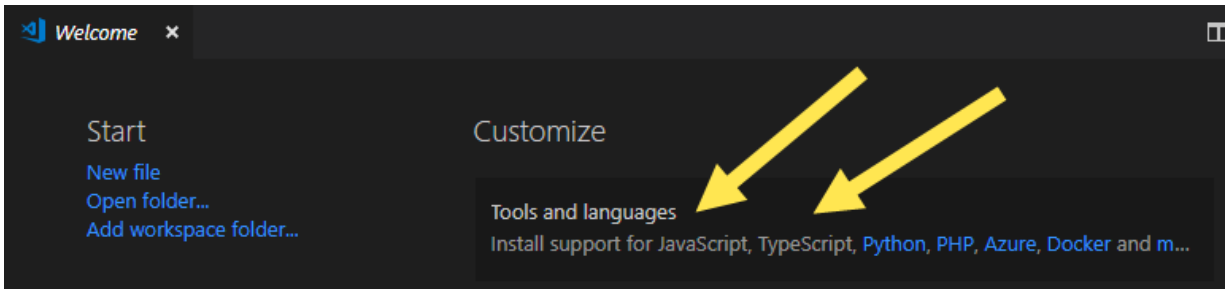


Figura 2.3: Intalando suporte ao TypeScript e JavaScript.

2.2 Testando a instalação do ambiente com um projeto básico

O Ionic fornece diversos modelos (templates) de projetos que podemos utilizar durante o processo de criação e você pode obter uma lista completa deles executando a instrução a seguir no terminal. Observe que são exibidas opções não só para o Angular.

```
ionic start --list
```

Caso você receba uma mensagem de erro relacionada aos direitos de acesso, será preciso você acessar o PowerShell como administrador e então executar a instrução a seguir, que possibilitará a atribuição de poderes ao usuário.

```
Set-ExecutionPolicy RemoteSigned
```

Vamos trabalhar a criação de um projeto básico (`blank`), mas recomendo que você teste a criação de projetos para todos os templates disponíveis, pois isso poderá trazer a você uma visão de alguns recursos que poderão ser utilizados em suas aplicações. Durante o desenvolvimento de exemplos do livro, criaremos as funcionalidades oferecidas por alguns deles.

Para a criação do projeto, recomendo que criemos uma pasta/diretório na raiz de seu disco e que executemos nela, estando no terminal, a instrução a seguir. Este processo pode demorar um pouco, pois depende de sua conexão com a internet, uma vez que o `Ionic CLI` realizará o download do template desejado. Veja na instrução que `capitulo02` é o nome do projeto, `blank` é o template que será utilizado, `type` define o `angular` como engine de templates e, no início, `start` define que o projeto deve ser criado. A execução dessa instrução poderá demorar um pouco. Como a partir da versão 5, o Ionic tem como padrão o Capacitor, será com base nele que o projeto será criado. Caso queira usar o Cordova, complemente a instrução a seguir com `--cordova` e, se quiser explicitar o Capacitor, complemente com `--capacitor` **.**

```
ionic start capitulo02 blank --type=angular
```

Logo no começo da execução do processo disparado pela instrução anterior, caso tenha escolhido o Cordova, uma orientação sobre a recomendação ao Capacitor, ao invés do Cordova, é exibida. Não se preocupe com avisos que possam surgir durante a execução da instrução anterior.

Uma mensagem de criação da pasta para o projeto será exibida. Caso você já tenha a pasta, será perguntado se deseja sobrescrever a que existe. Caso responda negativamente, a criação do projeto é interrompida. Durante a criação do projeto, algumas mensagens, orientações e sugestões serão exibidas. Ao final, uma pergunta sobre a instalação do `Ionic Pro SDK` é realizada. Responderemos com `N` neste momento. Observe que durante a instalação é informado que um repositório local do `Git` é criado. Poderíamos ter utilizado, na instrução de criação do projeto, o flag `--no-git` para que isso não ocorresse.

Você notou, na instrução para a criação do projeto, a definição do tipo como Angular (`--type=angular`)? Pois bem, como dito no capítulo anterior, o Ionic tem uma forte dependência com o Angular, mas agora as aplicações poderão fazer uso do `Vue` e `React` . Mas o que são esses artefatos?

Bem, vamos falar do Angular. Segundo seu próprio site (https://angular.io/), ele é uma plataforma que auxilia no desenvolvimento de aplicações web com foco nas `Simple Page Applications` , que são portais que possuem uma única URL e toda a informação dinâmica dela é configurada por meio de CSS e JavaScript. É uma ferramenta muito boa, poderosa, com muitos recursos e que vale a pena um estudo, principalmente, em nosso caso, pelo fato do Ionic fazer uso destes recursos. Neste livro, nosso foco é o Ionic, mas sempre que formos utilizar recursos vindos do Angular, manifestarei essa situação.

Pode ser que seja perguntado à você sobre criar ou não uma conta Ionic. É gratuita, mas fica a seu critério.

Para obtermos informações em relação ao Ionic, em sua pasta do projeto criada anteriormente, no terminal, digite a instrução a seguir. É a mesma que comentamos anteriormente.

```
ionic info
```

Você terá acesso a diversas informações sobre o ambiente e recursos utilizados no projeto Ionic. Em meu caso, em relação a versão apresentada para o Ionic/Angular foi a 5.9.2 e é para ela que implementaremos todo código aqui no livro. Não se preocupe com avisos que possam aparecer referente à pacotes Android e iOS. Logo resolveremos isso.

Em relação à execução do aplicativo Ionic em plataformas diferentes, tem-se o `Ionic Native`, que é uma biblioteca de plugins que possibilita de maneira simplificada a adição de funcionalidades nativas à plataforma em execução de aplicações móveis e, por isso, é bom termos a versão atualizada em nossos projetos.

Caso você queira fazer um backup da sua aplicação, você pode retirar a pasta `node_modules`, criada dentro de `capitulo02`, pois ela é imensa e pode ser gerada automaticamente por meio do NPM. Quer testar? Crie uma pasta chamada `teste`, no mesmo nível de `capitulo02` e copie para ela todo o conteúdo de `capitulo02`, com exceção da `node_modules`. Acesse a nova pasta pelo terminal e execute a seguinte instrução e, ao término, verifique que a pasta `node_modules` foi criada. Se quiser, pode apagar a pasta `node_modules` do `capitulo02` e executar a mesma instrução para instalar as dependências. Saiba que se utilizar o `Git`, esta pasta é totalmente ignorada.

```
npm install
```

Mas o que é o conteúdo da pasta `node_modules`? Ainda, o que são as dependências de que estamos falando de maneira constante neste início de livro? Você certamente notou que, para utilizarmos o Ionic, tivemos que instalar alguns recursos e que, ao criarmos um projeto, muitos recursos específicos para um projeto Ionic foram baixados e

disponibilizados na pasta `node_modules`. Estes recursos são vistos como dependências, sem as quais sua aplicação não pode ser desenvolvida e executada. Reforçando, de maneira mais direta, aplicações Ionic dependem muito de outras ferramentas, que podem ser aplicações, componentes e plugins. Veremos e trabalharemos essas dependências durante toda a leitura do livro.

Agora que já temos nosso projeto criado, vamos testá-lo? Acesse a pasta do projeto, que, seguindo nosso exemplo, deve se chamar `capitulo02`. Você pode acessá-la digitando `cd .\capitulo02\` no Windows PowerShell. Estando na pasta, digite a instrução a seguir.

```
ionic serve
```

Você poderá notar a exibição de algumas informações ao executar a instrução anterior. As informações iniciais se referem a como acessar a aplicação. Note que a primeira instrução executada é uma chamada ao `Angular CLI`, conhecido como `NG`. Na sequência, uma instância de seu navegador padrão será aberta, apresentando a aplicação, tal como é mostrado na figura a seguir. Para registro, eu tenho utilizado o Chrome como navegador padrão, por ser, de acordo com a comunidade, o navegador que está sempre atualizado com as tecnologias web.

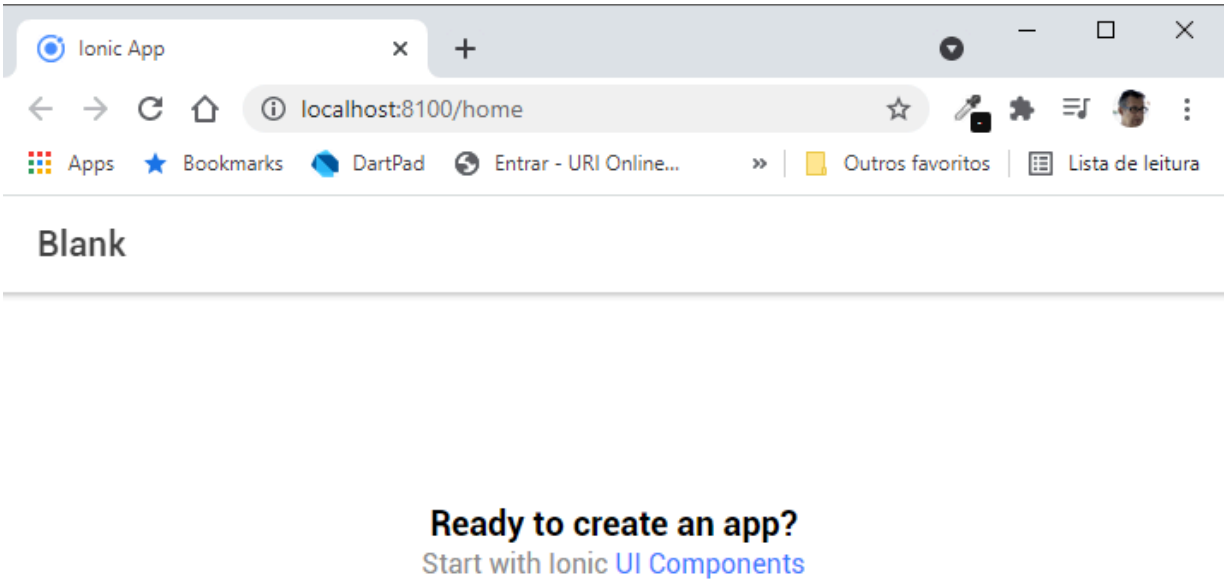


Figura 2.4: Execução básica da aplicação.

Olhando para a figura anterior, ou para seu navegador, você pode se perguntar, mas é assim que a aplicação executará em meu dispositivo móvel? Vamos por partes. Lembre-se de que as aplicações Ionic são Webview, ou seja, são executadas em um ambiente de um navegador web. Então, até o momento, está tudo certo com a execução da aplicação.

Retorne ao terminal e veja que várias instruções apareceram nele. São todas informações de LOG geradas pela preparação e execução da aplicação. Mas vamos verificar agora a execução da aplicação nos pseudoemuladores, que comentei no capítulo anterior. No terminal, pressione `CTRL-C` para interromper a execução. Após isso, no prompt, digite a instrução a seguir. Na primeira vez que executei, travou, precisei interromper (`CTRL-C`) e executar novamente.

```
ionic serve --lab
```

O flag `--lab` indicará ao Ionic que a execução da aplicação, embora em um navegador, deve ser apresentada em um ambiente de dispositivo móvel. Ao executar, ainda é possível que o Ionic retorne a você a informação de que não tem, nas dependências do projeto, o pacote `@ionic/lab` e se ofereça para instalá-lo. Vamos confirmar com `Y` caso isso ocorra. Após todo o processo de execução e instalação do pacote, o navegador é novamente aberto, agora exibindo a aplicação nos pseudoemuladores.

Permita-me apresentar um erro que pode acontecer com você quando for executar a instrução que habilita os pseudoemuladores. Em um determinado momento, quando executava a instrução anterior, nada aparecia no navegador. Parti para a inspeção de código do Chrome pressionando `F12` ou clicando com o botão direito do mouse em qualquer lugar da página e escolhendo a opção `Inspecionar`. A janela do navegador se divide e, no lado direito, é exibida uma poderosa ferramenta para depuração. Cliquei na guia `Console` e vi um erro com a descrição `Failed to load resource the server responded with a status of 404 (not found)`. Interrompi a execução do Ionic CLI e, na pasta do projeto, digitei `npm install` para forçar o download das dependências dos módulos node. Lembra que falei dessa instrução um pouco antes no texto? Após a instalação, executei novamente a instrução `ionic serve --lab` e tudo voltou a funcionar. Em outro caso, onde essa estratégia não funcionou, eu forcei a carga da página no navegador com `CTRL+F5` e tudo funcionou. Veja que essa segunda é mais simples e deve ser a primeira a ser tentada.

Após esse comentário, vamos ao navegador aberto. Veja a figura a seguir. Talvez apareça para você o pseudoemulador do Windows, mas eu preferi tirá-lo por meio do link `Platforms`. Essa opção se deu pela dificuldade de encontrar dispositivos Windows para testes físicos. Recomendo que deixe sempre apenas um pseudoemulador ativo, é mais rápido. Habilite mais de um apenas quando precisar ver a interface que o usuário visualizará.

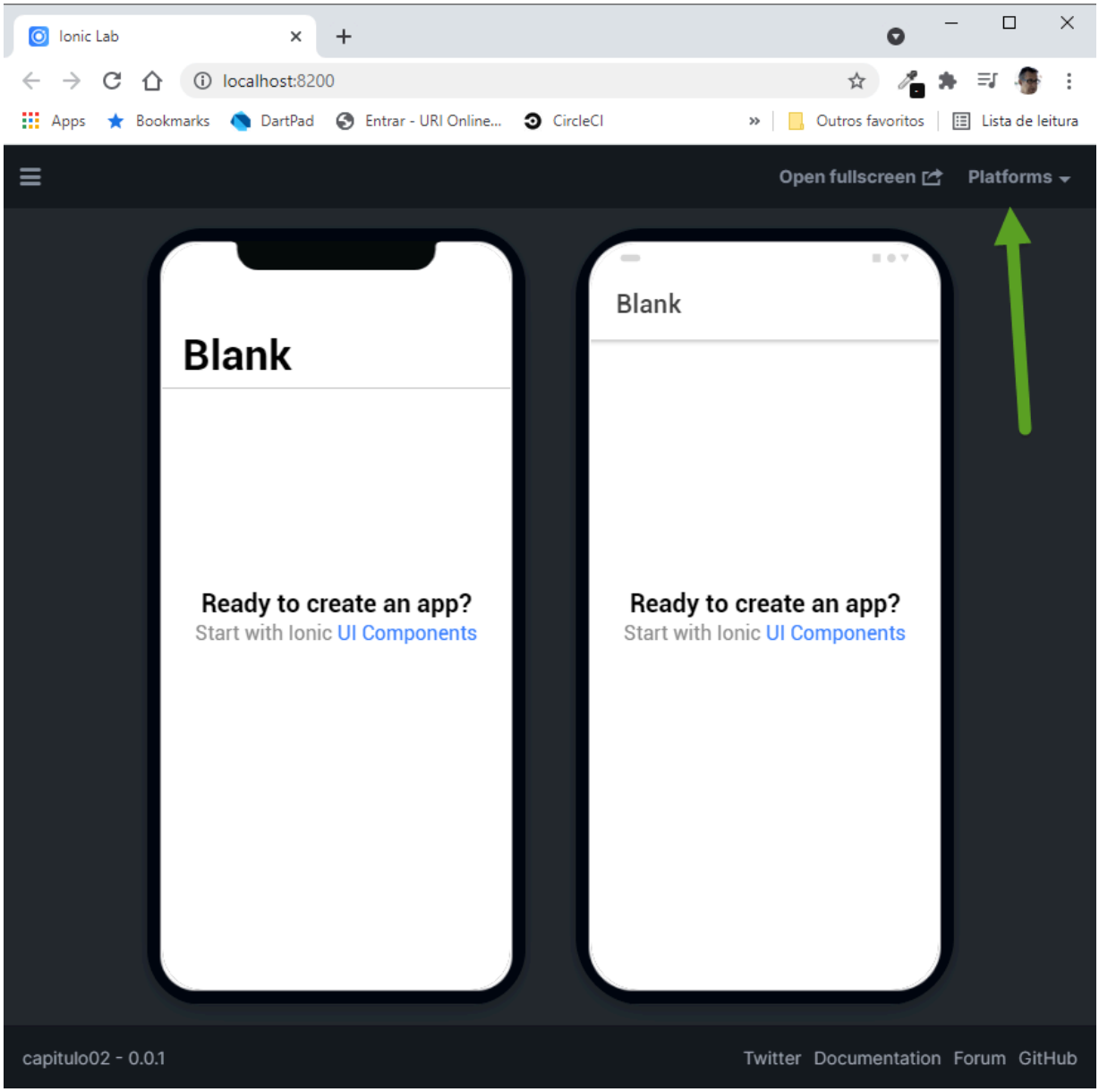


Figura 2.5: Execução da aplicação nos pseudoemuladores.

2.3 Execução de forma nativa no iOS

Lembra-se de que comentamos sobre aplicações Ionic poderem ser híbridas? Isso é possível por meio do uso de frameworks que permitem a comunicação de nossa aplicação com o SDK Nativo.

Essas aplicações são chamadas de `Native Bridge`. Vamos então testar a execução em emuladores e dispositivos reais? Vamos começar com o iOS, que tem a necessidade de termos nosso projeto sendo desenvolvido em um equipamento Mac. Mesmo que você não vá testar sua aplicação em um Mac, peço que leia toda esta seção, por ter procedimentos que serão importantes também para Android.

Como dito, o Ionic se integra a duas plataformas que possibilitam o acesso nativo, são elas o `Capacitor` (https://capacitor.ionicframework.com/) e o `Cordova` (https://cordova.apache.org/). Até a versão 4 do Ionic, o Cordova era o padrão, tendo o Capacitor como opção. A partir da versão 5, a recomendação do Ionic é o Capacitor, tendo o Cordova como opção, mas focaremos o livro no Capacitor.

Precisamos realizar algumas configurações para a execução do projeto. Vamos então abri-lo no Visual Studio Code. É importante saber que o ambiente de trabalho de um projeto neste IDE é a pasta onde o projeto está. Desta maneira, precisamos abrir essa pasta no VSC. Se você está com a guia `Welcome` aberta, clique no atalho `Open Folder` ou, se preferir, acesse o menu `File -> Open Folder` e localize a pasta para abri-la. Veja a figura a seguir, que apresenta o ambiente com a pasta aberta.

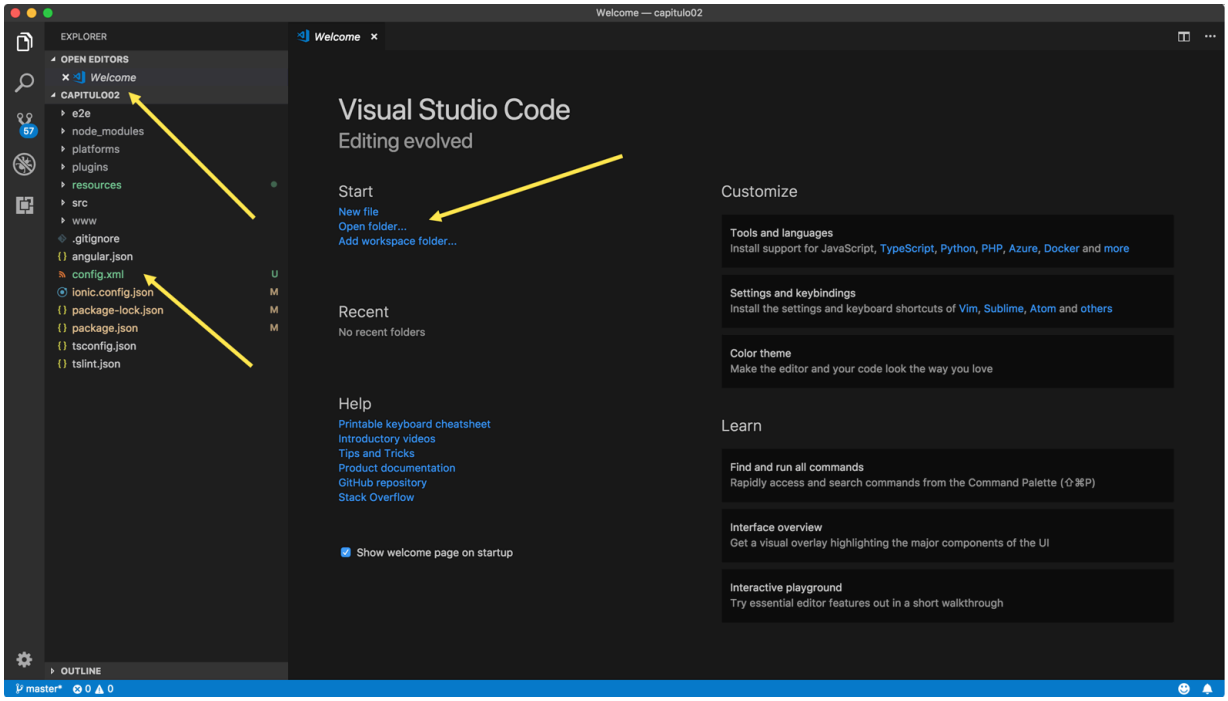


Figura 2.6: Visual Studio Code com a pasta do projeto aberta.

Com a pasta aberta, selecione o arquivo `capacitor.config.json` na raiz do projeto para que o VSC o abra em seu editor de código. Precisamos alterar o atributo `appId` e `appName`. O valor que devemos ter neste no `appId` deve respeitar uma notação chamada `Reverse domain name notation`, que traz o domínio da aplicação iniciando pelo fim. Por exemplo, teremos nosso domínio chamado `casadocodigoionic.com.br`. Assim, nosso `appId` será `br.com.casadocodigoionic` e em `appName` eu utilizei `Capitulo 02` apenas para ilustrar.

Se por algum acaso acontecer de o arquivo `capacitor.config.json` ser apagado, para que ele seja novamente criado, você precisa executar `npx cap init` para sua recriação.

Para que possamos executar nossa aplicação iOS no Mac, precisamos do XCode instalado, pois é a ferramenta oficial da Apple para desenvolvimento. Se você não tiver o aplicativo em seu Mac, precisará instalá-lo pela `App Store`. No momento da escrita deste livro, ele estava na versão 13.1 (13A1030d).

A preparação de nosso ambiente para o uso do Capacitor, no Mac, demanda um conjunto de pré-requisitos e dependências. A primeira delas se refere à instalação da dependência `CocoaPods`, que é um gerenciador de dependências para projetos `Swift` e `Cocoa Objective-C`. Não entrarei em detalhes sobre isso por não fazer parte do escopo do livro. Sendo assim, no terminal de seu Mac, digite as instruções a seguir.

```
// Instalação do Cocoapods. A versão neste momento é a 1.10.2
sudo gem install cocoapods

// Instalação Xcode Command Line tools, se necessário
xcode-select --install

// Atualização do repositório local do CocoaPods
pod repo update
```

Para que você se torne um desenvolvedor certificado pela Apple e possa fazer parte do `Apple Developer Program`, existe uma série de tramitações que devem ser seguidas e que estão muito bem documentadas no portal da Apple. Entretanto, para fazer um deploy como um desenvolvedor "simples" e sem custo, existe um processo mais simples de ser realizado, e é ele que veremos aqui.

O primeiro requisito é ter um Id Apple. Se não tem, obtenha um em https://appleid.apple.com/account#!&page=create/. Em seguida, em seu Mac, acesse o XCode e acesse o `XCode Menu -> Preferences`. Na categoria `Accounts`, clique na base esquerda, no botão `+` para adicionar seu Id Apple. Sua senha será solicitada neste momento e a janela a seguir é exibida.

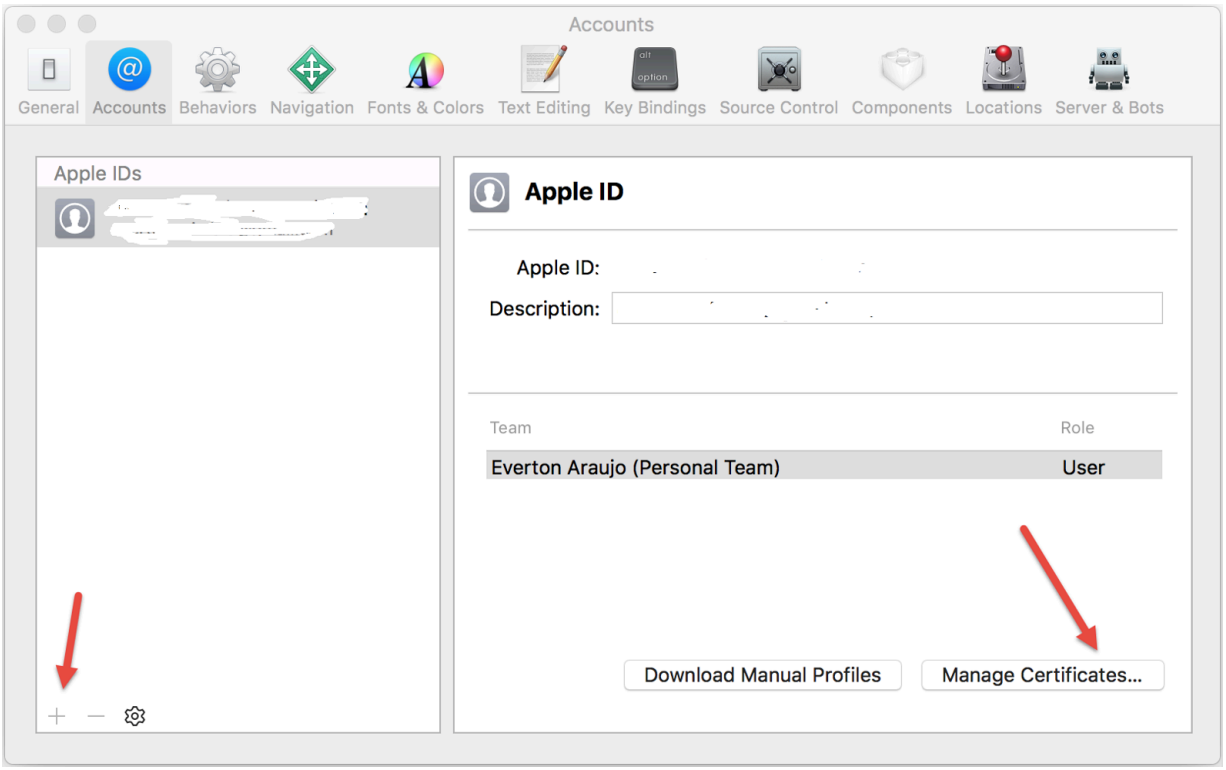


Figura 2.7: Criando uma conta no XCode.

Com a conta criada, precisaremos criar uma `Signing Identity`. Clique no botão `Manage Certificates...` e a próxima janela é exibida. Clique no botão `Create` e selecione `iOS Development`. Após estes passos, você pode, se quiser, fechar o XCode.

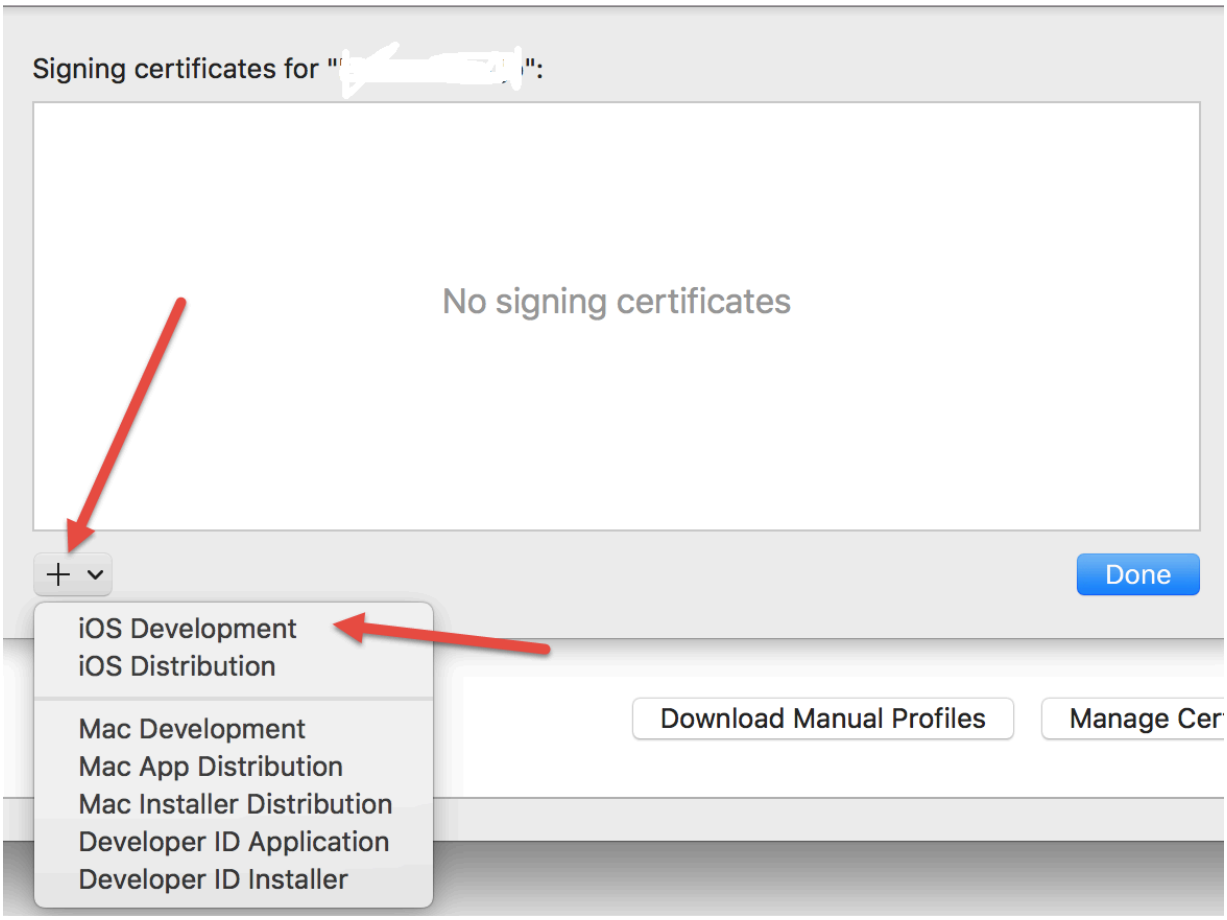


Figura 2.8: Criando uma Signing Identity.

Após termos a configuração e ferramentas necessárias disponíveis em nosso equipamento, precisamos dar sequência em instruções para preparar nosso projeto e abri-lo no XCode. Veja na sequência.

```
// Adição da plataforma iOS ao projeto
ionic capacitor add ios

// Construção e abertura do projeto iOS no XCode
ionic capacitor build ios
```

Você pode optar por abrir o projeto já com o XCode aberto. O que devemos fazer é abrir a pasta `platforms/ios`, que temos dentro de nossa pasta `capitulo02`. Esta pasta foi gerada quando adicionamos o iOS ao projeto.

Com o projeto aberto, selecione o nó principal (`App`) e, na guia que se abre, verifique se os campos `Display Name` e `Bundle Identifier` , na categoria `Identity` , estão como configuramos no VSC. Se não estiverem, ajuste.

Em seguida, na categoria `Signing & Capabilities` , marque o checkbox `Automatically manage signing` e selecione o `Team` , que você criou. Para finalizar esta etapa, na categoria `Deployment info` (na guia `General` , selecione a versão alvo para sua aplicação. Em meu caso, selecionei a mais recente, que era a 15.0. Com essa seleção, estamos dizendo que os dispositivos que poderão executar nossa aplicação deverão estar atualizados, no mínimo, com o iOS 15.0. Se você quiser que versões anteriores possam executar seu aplicativo, basta selecioná-las. Lembre-se de que quanto menor a versão alvo, menos recursos atuais da plataforma estarão disponíveis e quanto mais recente a versão, menos dispositivos desatualizados poderão executá-la.

Com as configurações realizadas, você pode executar sua aplicação. No topo da janela do XCode, existe um botão de `play` e, ao lado dele, o nome da aplicação seguido pelo dispositivo onde sua aplicação executará. Essa localização pode ser diferente dependendo da versão do XCode que utiliza, mas, se clicar diretamente no `play` , um simulador será aberto. Na listagem de dispositivos, a princípio, são exibidos simuladores, mas se você tiver um dispositivo conectado ao Mac, ele também aparecerá na listagem. Para começarmos, escolha um simulador e execute. O processo de deploy (implantação) ocorrerá, e a aplicação executará.

Vamos ao dispositivo físico agora? Conecte-o ao Mac, selecione-o na lista de dispositivos no XCode e execute a aplicação. Se neste momento uma janela de erro aparecer em seu dispositivo, informando-o para verificar configurações de confiança, não se preocupe, isso ocorre na primeira execução de uma aplicação via o mecanismo que estamos utilizando, o de depuração. Realize a configuração indicada pelo dispositivo, que é bem simples, e execute novamente sua aplicação. Em meu iPhone, eu acessei `Ajustes->Geral->Gerenciamento de Dispositivo->Selecionei meu ID Apple->Confirmar`

`confiança` . Se tudo deu certo, a aplicação será executada em seu dispositivo. Lembre-se de encerrar a execução da aplicação no XCode ou no próprio dispositivo.

Se formos realizar alguma alteração no projeto, as atualizações não são aplicadas diretamente na aplicação que executamos no emulador/dispositivo. Precisamos executar novamente a instrução `ionic capacitor build ios` e voltar ao XCode e executar novamente a aplicação. Não é preciso fechar o XCode e nem o projeto nele para isso. Logo veremos a execução da aplicação diretamente pelo terminal.

E se realizarmos alterações em nosso projeto Ionic? Precisamos novamente realizar o processo de construção que executamos anteriromente?

A boa notícia é que o Capacitor traz também a funcionalidade de `Live Reload` . Isso nos permite executar a aplicação no dispositivo, realizarmos alterações em nosso projeto e vermos essas atualizações serem realizadas no aplicativo em execução, sem ter que utilizar o XCode.

A execução com o Capacitor com `Live Reload` é simples. Veja o código a seguir, que exibirá uma lista de dispositivos e emuladores disponíveis para a execução, ficando para você a responsabilidade dessa escolha. Caso na execução das instruções a seguir, você receba um erro `ERR_CONNECTION_REFUSED` , insira, ao final da instrução o argumento `--external` . É uma recomendação do próprio Ionic.

```
ionic capacitor run ios -l
```

Caso você queira maior agilidade, dizendo já na instrução em qual dispositivo você deseja executar seu aplicativo, podem ser utilizadas as instruções a seguir.

```
// Para obter uma relação dos emuladores e dispositivos disponíveis
ionic capacitor run ios --list
```

```
// Para executar em um específico dispositivo
ionic capacitor run ios -l --target="<TARGET ID DO DISPOSITIVO>"
```

Com a aplicação em execução, modifique alguma coisa no `home.page.html` (pelo Visual Studio Code), grave e observe seu dispositivo. A alteração apareceu? Legal, não é?

Diversas outras opções são disponibilizadas pelo Capacitor. Caso se interesse, recomendo a verificação dessas opções na documentação oficial disponibilizada em https://ionicframework.com/docs/cli/commands/capacitor-run.

2.4 Execução de forma nativa no Android

O procedimento para utilização do Capacitor para o Android é semelhante ao que vimos para iOS. Precisamos da mesma configuração no `capacitor.config.json` e da execução das instruções a seguir para termos o projeto agora sendo aberto no Android Studio. Caso lhe seja perguntado sobre atualizações no projeto Android, responda sempre com não.

```
// Adição da plataforma iOS ao projeto
ionic capacitor add android

// Construção e abertura do projeto iOS no XCode
ionic capacitor build android
```

Vamos executar agora a aplicação Android diretamente do terminal, proporcionando o Live Reload para as alterações realizadas no código Ionic. Essa execução é igual ao que fizemos para o iOS, mudando apenas a plataforma. Veja a instrução a seguir.

```
ionic capacitor run android -l
```

Caso você queira maior agilidade, dizendo já na instrução em qual dispositivo você deseja executar seu aplicativo, podem ser utilizadas

as instruções a seguir.

```
// Para obter uma relação dos emuladores e dispositivos disponíveis
ionic capacitor run android --list

// Para executar em um dispositivo específico
ionic capacitor run android -l --target="<TARGET ID DO DISPOSITIVO>" --
host=0.0.0.0
```

Com a aplicação em execução, modifique alguma coisa no `home.page.html` (pelo Visual Studio Code), grave e observe seu dispositivo. A alteração apareceu? Legal, não é?

Diversas outras opções são disponibilizadas pelo Capacitor. Caso se interesse, recomendo a verificação dessas opções na documentação oficial disponibilizada em https://ionicframework.com/docs/cli/commands/capacitor-run.

Para executarmos a aplicação em um dispositivo Android, precisamos habilitar as opções para o desenvolvedor (*Developer Options*). Para isso, no dispositivo, vá em `Configurações -> Sobre o dispositivo -> Informações de Software` e clique sete vezes sobre o `Número de compilação` (*build number*). Após isso, retorne para `Configurações` e clique em `Opções do Desenvolvedor` (*Developer Options*) e habilite a opção `USB Debugging`. Se essas opções não aparecerem para seu dispositivo, você precisará buscar na documentação dele como acessá-las. Normalmente, nos sites de pesquisa, ao digitar as palavras-chaves `android ativar modo desenvolvedor <fabricante-e-modelo-dispositivo>`, as orientações necessárias já são trazidas.

Com seu dispositivo liberado para testes e devidamente conectado ao seu computador, feche o emulador e, no Android Studio, selecione seu dispositivo na lista disponibilizada ao lado do `play` e execute sua aplicação. Bem mais simples que no iOS, certo?

2.5 Depuração

Depurar uma aplicação é tão importante quanto a criação de código. Entretanto, depurar um código TypeScript, ou JavaScript, em tempo de execução não é algo trivial pela característica das tecnologias empregadas. Temos uma ferramenta poderosa no console do navegador. Mas e quando utilizamos um dispositivo? Também fazemos uso do console, que, no caso do Chrome, é de fácil acesso, mas também há disponibilidade dele para o Safari.

Para termos o console do dispositivo acessível, utilizando o navegador Chrome, precisamos informar a URL chrome://inspect/#devices/. Os dispositivos físicos conectados a seu equipamento ou via rede aparecerão na página aberta, assim como a aplicação em execução no emulador. Precisamos então clicar no link `Inspect`, que aparece, e nossa janela de console se tornará disponível. Isso ajuda e muito, por isso, guarde bem essa informação.

Conclusão

Chegamos ao final deste segundo capítulo. Conseguimos identificar as ferramentas que utilizaremos, realizamos as instalações delas e as testamos em um projeto simples. Foi possível também executarmos a aplicação desenvolvida em dispositivos físicos e emuladores específicos do iOS e Android, por meio do Capacitor, o que nos viabiliza o uso nativo de recursos dos aparelhos. Pudemos verificar que existe sempre a possibilidade de ocorrência de alguns erros/problemas, mas todos são contornáveis. Muita coisa interessante foi vista, e estamos apenas no segundo capítulo. Vamos pegar um fôlego antes de começarmos o terceiro?

Capítulo 3
A primeira aplicação Ionic

No capítulo anterior, criamos uma aplicação básica, com vistas apenas a testar o ambiente necessário para o desenvolvimento e a execução de aplicações Ionic.

A aplicação que iniciaremos aqui será responsável pela interface com o usuário para manipulação de dados relacionados a clientes.

Começaremos o capítulo apresentando a estrutura básica de um projeto criado pelo Ionic CLI, os componentes básicos implementados no `home` e de como ocorre a execução da aplicação. Na sequência, criaremos nosso template para clientes com alguns componentes para a interação com o usuário.

A partir deste capítulo, o foco em execução se concentrará no Android, mas, caso alguma intervenção específica para o iOS seja necessária, ela será trazida.

3.1 A estrutura básica de um projeto Ionic

Vamos criar um novo projeto para trabalharmos neste capítulo. Dessa maneira, em sua pasta de projetos Ionic, insira no terminal, como sugestão, a seguinte instrução. Observe que optei pelo não uso do GIT neste projeto. Após a adição das plataformas, verifique a instalação do pacote `@ionic-native` , necessário para acesso à componentes que precisaremos.

```
// Lembre-se de estar na pasta raiz de seus projetos
ionic start capitulo03 blank --type=angular --no-git

// Adição das plataformsa iOS e Android ao projeto.
ionic capacitor add ios
ionic capacitor add android
```

Após a criação, vamos abrir a nova pasta (capitulo03) no Visual Studio Code. Veja a estrutura criada na figura a seguir.

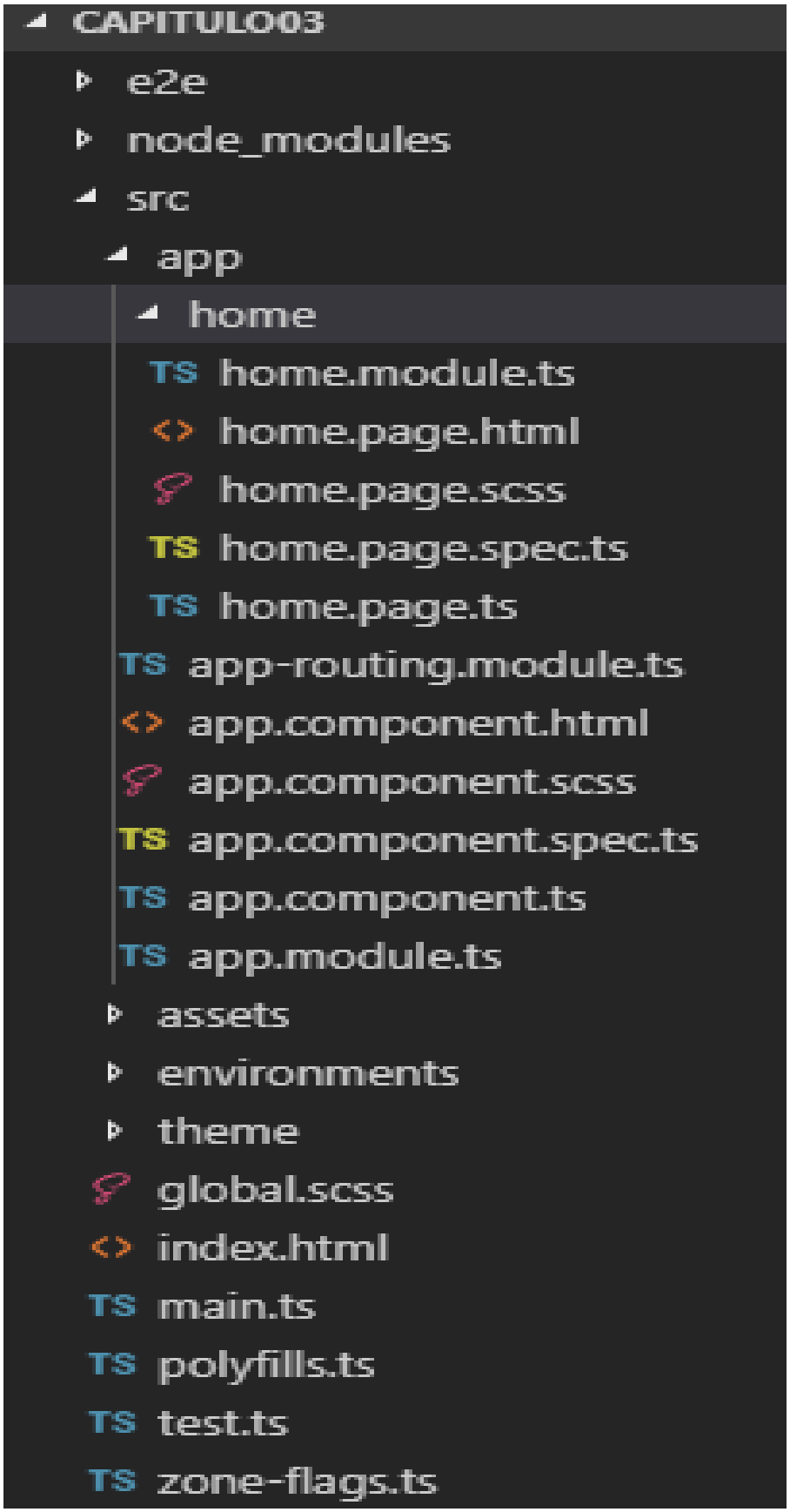

CAPITULO03
e2e
node_modules
src
app
home
home.module.ts
home.page.html
home.page.scss
home.page.spec.ts
home.page.ts
app-routing.module.ts
app.component.html
app.component.scss
app.component.spec.ts
app.component.ts
app.module.ts
assets
environments
theme
global.scss
index.html
main.ts
polyfills.ts
test.ts
zone-flags.ts

Figura 3.1: Estrutura criada para o projeto.

A estrutura adotada por nossa aplicação Ionic 6 segue a estrutura básica de uma aplicação criada em Angular, que foi o framework para templates que escolhemos. Note que temos a pasta `src` na raiz da pasta do projeto. Dentro dela, temos outra, chamada `app`, que é onde ficará o código que implementaremos para nossa aplicação. A fala sobre Angular em alguns pontos do livro terá o foco de apenas demonstrar a relação entre os frameworks. Você não precisará aprender o Angular para nossas atividades, mas é uma ferramenta cujo estudo recomendo.

Por convenção, o Angular organiza seus artefatos (arquivos) em pastas e, dentro delas, que por enquanto podemos chamar de contextos, os arquivos que são relacionados a elas. A estes contextos, o Ionic dá o nome de componentes ou módulos, dependendo de como os artefatos forem criados. Veremos isso quando começarmos a trabalhar nos códigos. Essas pastas de componentes ficam sempre dentro de `src/app`, como é o caso de `home`.

Abra a pasta `home` e veja que temos seis arquivos, todos com o nome composto por `<nome-contexto/pasta/componente>.<tipo-do-artefato>.<tipo-do-arquivo>`. Observe que desses arquivos, quatro têm o tipo de artefato `page`, pois são eles que compõem uma página, cada um com sua finalidade e tipo de arquivo. Temos:

- `html`, que se refere ao template (modelo) que será renderizado pela aplicação.
- `scss` são arquivos que contêm os CSSs relacionados ao template em questão. SCSS é um arquivo que contém código Sass, que é uma linguagem com recursos para manipulação de estilos.
- `ts` conterá código TypeScript, que serão `transpilados` (traduzidos) para JavaScript.
- `spec.ts` mantém códigos de testes para o componente em questão. Nós não trabalharemos com testes neste livro.

Abra o arquivo `home.page.html` clicando no nome dele, pois é assim que o Visual Studio Code abre um arquivo para edição. Trago para discutirmos o código existente neste arquivo, que já conhecemos do capítulo anterior.

```
<ion-header>
  <ion-toolbar>
    <ion-title>
      Ionic Blank
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content [fullscreen]="true">
  <ion-header collapse="condense">
    <ion-toolbar>
      <ion-title size="large">Blank</ion-title>
    </ion-toolbar>
  </ion-header>

  <div id="container">
    <strong>Ready to create an app?</strong>
    <p>Start with Ionic <a target="_blank" rel="noopener noreferrer"
href="https://ionicframework.com/docs/components">UI Components</a></p>
  </div>
</ion-content>
```

Todo o código que temos são tags que representam componentes do Ionic e, se necessário, usamos também as tags HTML. O arquivo HTML, apresentado na listagem, é chamado, pela convenção do Ionic/Angular, de template/modelo. Podemos ainda criar nossos componentes e utilizar componentes de terceiros na criação de nossos templates.

3.2 Os componentes básicos do template home

No VSC, ative um terminal clicando no menu `Terminal -> New Terminal`. Existe um conjunto de teclas de atalho para isso que você pode utilizar, o `CTRL + SHIFT + '`.

Por praticidade, realizarei as execuções com mais frequência no próprio navegador. Como estamos começando no livro, vou trazer novamente aqui a instrução que realizará essa execução. Veja o código na sequência e, após ele, os componentes identificados na imagem.

```
ionic serve --lab
```

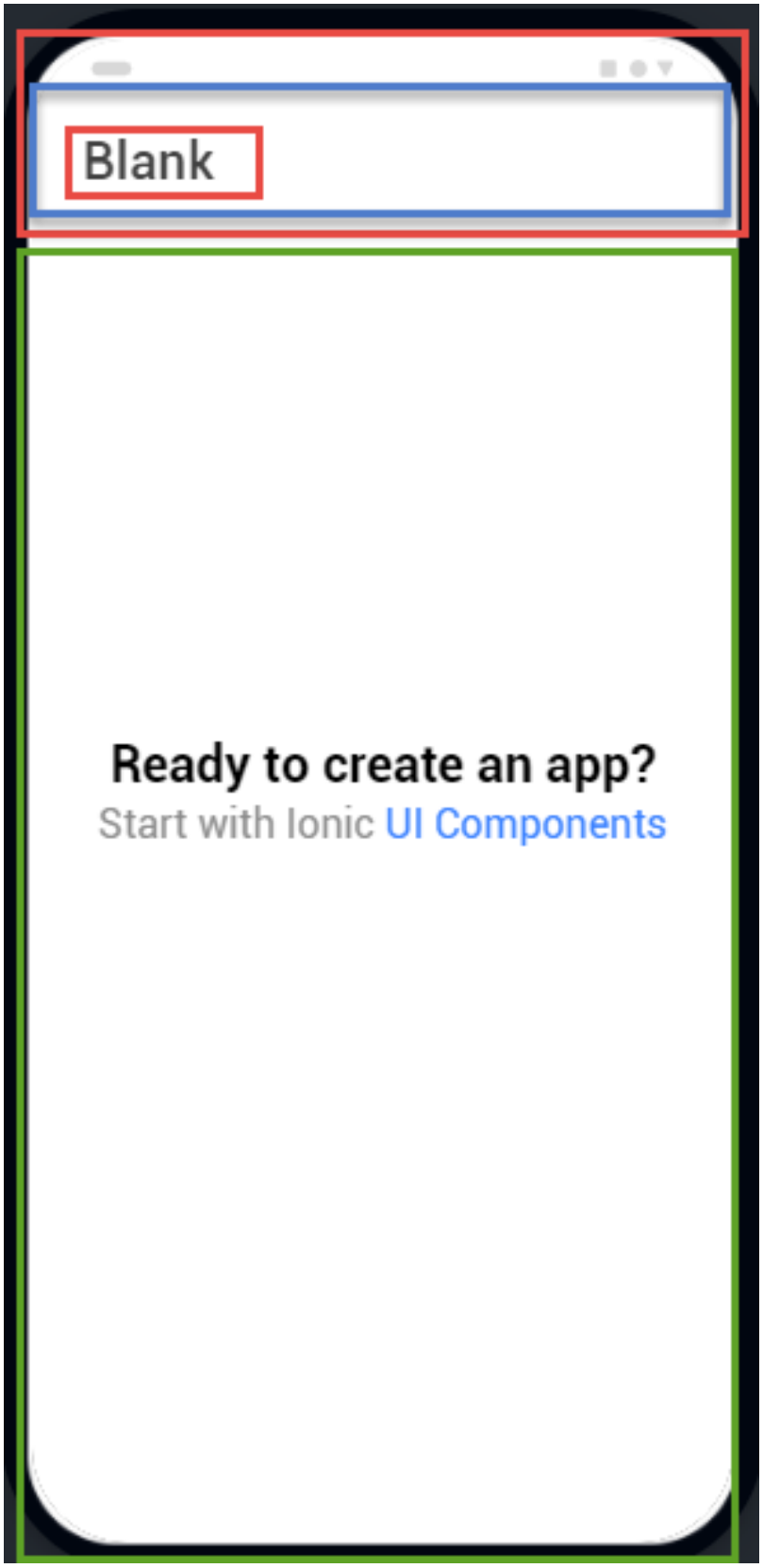

Blank
Ready to create an app?
Start with Ionic UI Components

Figura 3.2: Identificando os componentes no template em execução.

Observe o retângulo vermelho (o mais externo), ele representa o `<ion-header>` e é um componente obrigatório para todos os templates de página. Dentro deste componente, temos o `<ion-toolbar>`, representado pelo retângulo azul. Tente escrever algo dentro dele, antes do `<ion-title>`, e verifique a aplicação em execução. Ele escreve o que digitamos, mas sem respeitar as características das plataformas para o título da página. Por isso, fazemos uso do componente `<ion-title>` dentro da toolbar, pois, na hora de executar a aplicação, o Ionic respeitará cada plataforma. Todo o restante de nossa página, representado pelo retângulo verde, é a área de conteúdo e que é identificada pelo componente `<ion-content>`. A diretiva `padding` informa que será preciso realizar um espaçamento entre as bordas do template e seu conteúdo. A título de curiosidade, remova a diretiva e veja a diferença na execução.

Para o Ionic, a definição do template, pelo arquivo HTML, não é suficiente para que a página seja renderizada. Ela é um artefato que utiliza outros artefatos. Então, ela precisa de informações sobre como esses artefatos, e ela, serão renderizados. Veja o código de `home.page.ts` na sequência.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss'],
})
export class HomePage {
  constructor() {}
}
```

Observe que temos uma classe chamada `HomePage`, que é precedida por um `decorator` chamado `@Component`. Este decorator receberá um objeto JavaScript com três propriedades: a `selector` define como o componente será utilizado/referenciado na aplicação. Como nosso

componente tem escopo de página, essa propriedade poderia não ser configurada aqui; a `templateURL` aponta para o modelo que será utilizado para renderizar a classe. Veja que ela aponta para a página que discutimos há pouco; por fim, temos a propriedade `styleUrls`, que também define qual o arquivo, exclusivo ao template, que conterá código CSS que será utilizado para o template em questão. Decorators marcam elementos de código, neste caso, o `@Component()` precede uma classe chamada `HomePage`.

Outro ponto importante é a instrução `export` antes da definição da classe. É desta maneira que definimos que a classe poderá ser consumida por outros módulos e componentes.

Para que nosso componente `Home` esteja disponível para a aplicação, precisamos empacotá-lo em um módulo. Abra o arquivo `home.module.ts`. Para facilitar, na sequência trago este código.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { IonicModule } from '@ionic/angular';
import { FormsModule } from '@angular/forms';
import { HomePage } from './home.page';

import { HomePageRoutingModule } from './home-routing.module';


@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    HomePageRoutingModule
  ],
  declarations: [HomePage]
})
export class HomePageModule {}
```

Os imports trazem, para o contexto do artefato em questão, os módulos, plugins, componentes e qualquer tipo de artefato que seja

necessário para a classe (ou módulo). Observe que temos a definição da classe precedida de um `decorator` do Angular, o `@NgModule()`, que recebe um objeto JavaScript.

Note a propriedade `declarations` do `decorator`, que informa quais componentes/classes são declarados no módulo em questão. Informamos o `HomePage` entre colchetes, o que caracteriza um array.

O `HomePageRoutingModule` é o responsável pela navegação entre páginas e está implementado no arquivo `home-routing.module` cuja implementação podemos verificar na sequência.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomePage } from './home.page';

const routes: Routes = [
  {
    path: '',
    component: HomePage,
  }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class HomePageRoutingModule {}
```

No início do arquivo, após os `imports`, temos a definição da constante `routes`, que é um array de objetos JavaScript, onde a propriedade `path` refere-se ao caminho que define quando o módulo será requisitado e `component`, que define qual componente será retornado quando a rota definida em `path` for informada. Em nosso exemplo, quando a raiz de nossa aplicação for requisitada, o componente `HomePage` será renderizado.

A definição da classe `HomePageRoutingModule` é precedida também por um decorator `NgModule`. Veja que ele importa as rotas definidas em

`Routes` e essa importação se dá pelo método `forChild()`, que é recomendado para carregamento tardio (*lazy loading*) de submódulos, ou seja, o módulo será carregado apenas quando for requisitado.

3.3 A execução da aplicação

Com a explanação realizada para os componentes da página `home`, que é criada para todo projeto do modelo `blank`, podemos agora buscar uma compreensão de como ocorre a execução da aplicação.

Abra o arquivo `index.html`, que está na raiz da pasta `src`. No final do arquivo, dentro da tag `<body>`, note a existência de um componente Ionic chamado `<app-root>`.

O `<app-root>` é o componente de inicialização de uma aplicação Ionic. Abra o arquivo `main.ts`, que está também na pasta `src`. Observe as instruções a seguir, que estão no final do arquivo. Veja que é enviado `AppModule` para o método `bootstrapModule()`, encadeado ao `platformBrowserDynamic()`. Esses métodos inicializam a aplicação e indicam que ela começa com o módulo `AppModule`.

```
platformBrowserDynamic().bootstrapModule(AppModule).catch(err =>
console.log(err));
```

A maioria dos componentes do Ionic pertence a módulos, e um módulo pode conter um ou vários componentes. Os módulos podem ser vistos também como uma maneira de organizar os componentes.

Quando nossa aplicação for executada, já sabemos que ela buscará pelo arquivo `index.html` e que ele tem o componente `<app-root>`, que é o que inicializa a aplicação Ionic. Ele estará disponível para a página e aplicação por meio do código anterior e, no início do código, temos uma instrução que traz para o contexto de `main.ts` o módulo `AppModule`, que é:

```
import { AppModule } from './app/app.module';
```

É possível também verificar, no início do arquivo, vários outros imports, todos necessários para a inicialização da aplicação.

Mas e onde está o `AppModule`? Observe que, no código anterior, importamos `./app/app.module` para uma referência chamada `AppModule`. Vamos então abrir este arquivo, dentro de `src/app`. Veja no código do arquivo a chave `declarations`, com o valor `AppComponent`, e a chave `bootstrap`, como o valor `AppComponent`, que será o componente de inicialização do módulo.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouteReuseStrategy } from '@angular/router';

import { IonicModule, IonicRouteStrategy } from '@ionic/angular';

import { AppComponent } from './app.component';
import { AppRoutingModule } from './app-routing.module';

@NgModule({
  declarations: [AppComponent],
  entryComponents: [],
  imports: [BrowserModule, IonicModule.forRoot(), AppRoutingModule],
  providers: [{ provide: RouteReuseStrategy, useClass: IonicRouteStrategy
}],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Mas vamos então ao componente `AppComponent`. Abra o arquivo `app.component.html`. Viu que não é `page`, como no caso do `home`? Isso nos leva a deduzir que o arquivo é um template para um componente (uma tag) que será utilizada em outro componente ou página, como de fato vimos no arquivo `index.html`. Veja o código do template na sequência.

```
<ion-app>
  <ion-router-outlet></ion-router-outlet>
</ion-app>
```

No template, note que temos a utilização de um novo componente, o `<ion-app>` e, dentro dele, o `<ion-router-outlet>`, que veremos mais à frente no livro. Para este momento, precisamos saber apenas que é neste componente que as páginas serão renderizadas. O `<ion-app>` é o componente que define realmente que começamos com uma aplicação Ionic. Vamos abrir agora o arquivo `app.component.ts`. Logo no início, repare no decorator `@Component()`. Viu o valor de `selector`? É o `app-root`, que vimos no `index.html`. Viu só toda a dinâmica e burocracia? O código da classe está a seguir.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
  styleUrls: ['app.component.scss'],
})
export class AppComponent {
  constructor() {}
}
```

Para fecharmos a seção, vamos à uma breve descrição de todo o processo:

1. Executamos a aplicação.
2. O arquivo `index.html` é invocado em conjunto com a classe `main.ts`.
3. O `AppModule` é carregado, trazendo o `AppComponent` para a aplicação, que define o `<app-root>` e inicializa a aplicação Ionic com o `<ion-app>`.
4. Quando a aplicação está iniciada, a URL padrão para ela é apenas o endereço do servidor, ou seja, o `path` raiz definido no `home.module.ts`, e a página `home.page.html` é renderizada em conjunto com a classe definida em `home.page.ts`.

3.4 Criação da página de clientes

Vamos dar início à implementação de nossa aplicação implementando uma página que represente a entrada de dados do nosso cliente. Uma página (page) é um componente, e o Ionic CLI nos oferece um mecanismo para nos auxiliar na geração de componentes. Você pode, no terminal, digitar uma instrução e informar os dados que auxiliarão o Ionic CLI na criação do componente ou pode informar diretamente na linha de comando que deseja criar uma página. Veja essas instruções na sequência. Você pode abrir outro terminal, sem a necessidade de interromper a execução da aplicação, ok?

```
// Criação com pergunta/resposta dos componentes. Utilize a instrução a seguir.
ionic generate

// Criação automática de nossa página . Utilize essa
ionic generate page cliente/cliente-add-edit
```

Na segunda instrução, estamos informando que queremos criar uma página chamada `cliente-add-edit`, que deverá estar em uma pasta chamada `cliente/cliente-add-edit`.

Após a execução da instrução anterior, verifique as pastas criadas dentro de `src/app/pages`. Vamos agora codificar nosso template de componentes Ionic, o arquivo `cliente-add-edit-page.html`. Veja na sequência o código inicial para o template.

```
<ion-header>
  <ion-toolbar>
    <ion-title>
      Registro de CLIENTES
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content padding>
```

```
  <ion-input type="text" placeholder="Nome"></ion-input>
  <ion-input type="email" placeholder="Email"></ion-input>
  <ion-input type="tel" placeholder="Telefone"></ion-input>
  <ion-input type="number" placeholder="Renda" text-right></ion-input>
  <ion-input type="text" placeholder="Nascimento"></ion-input>
  <ion-button shape="round" color="primary" expand="block" padding>
      Confirmar
  </ion-button>
</ion-content>
```

Podemos ver, dentro de `<ion-content>`, diversas tags `<ion-input>`, que são responsáveis pelo componente de entrada de dados em forma de texto, e um `<ion-button>`, que renderizará um componente de botão.

Observe o atributo `type` e suas variações de valores para ele, de acordo com o dado que estaremos solicitando ao usuário. Veja o uso de `text`, `email`, `tel` e `number`, que são, respectivamente, texto, e-mail, telefone e números. Este atributo está diretamente relacionado ao tipo de teclado virtual que será exibido ao usuário quando estiver nos campos de entrada. Observe também o atributo `placeholder`, com um texto que será exibido na caixa de entrada, para auxiliar o usuário em relação ao dado que é preciso informar. Note a diretiva `text-right` para renda, para que o texto dentro do componente de entrada seja alinhado à direita.

O `<ion-button>`, em nosso exemplo, faz uso dos atributos `shape`, `color`, `expand` e da diretiva `padding`. O primeiro define a forma de renderização do botão, que está como `round` (cantos arredondados), sendo possível ainda o uso dos valores `full` e `block`. Para `color`, estamos fazendo uso de valores pré-criados e que podem ser vistos no arquivo `variables.css`, dentro da pasta `theme`, na raiz do projeto (src). O atributo `expand` define o padrão de preenchimento do componente na página, que em nosso caso ocupará todo o bloco disponível. Finalizamos o componente informando que deve ser aplicado o `padding` em todas as bordas.

Precisamos agora fazer com que nossa página seja renderizada, pois, ao executarmos o `ionic serve --lab`, veremos a página `home`. Vamos então abrir o arquivo `app-routing.module.ts`, que está dentro da pasta `src/app`. Para o momento, vamos alterar a rota para o caminho `''`, que está redirecionando a requisição para `home`. Altere então para que a rota fique de acordo com o código a seguir.

```
{ path: '', redirectTo: 'cliente-add-edit', pathMatch: 'full' }
```

Se você estiver com o `lab` aberto, recarregue a página (F5). Caso ela não renderize corretamente, será preciso interromper o `serve` e invocá-lo novamente. A figura a seguir apresenta o código anterior renderizado nos pseudoemuladores.

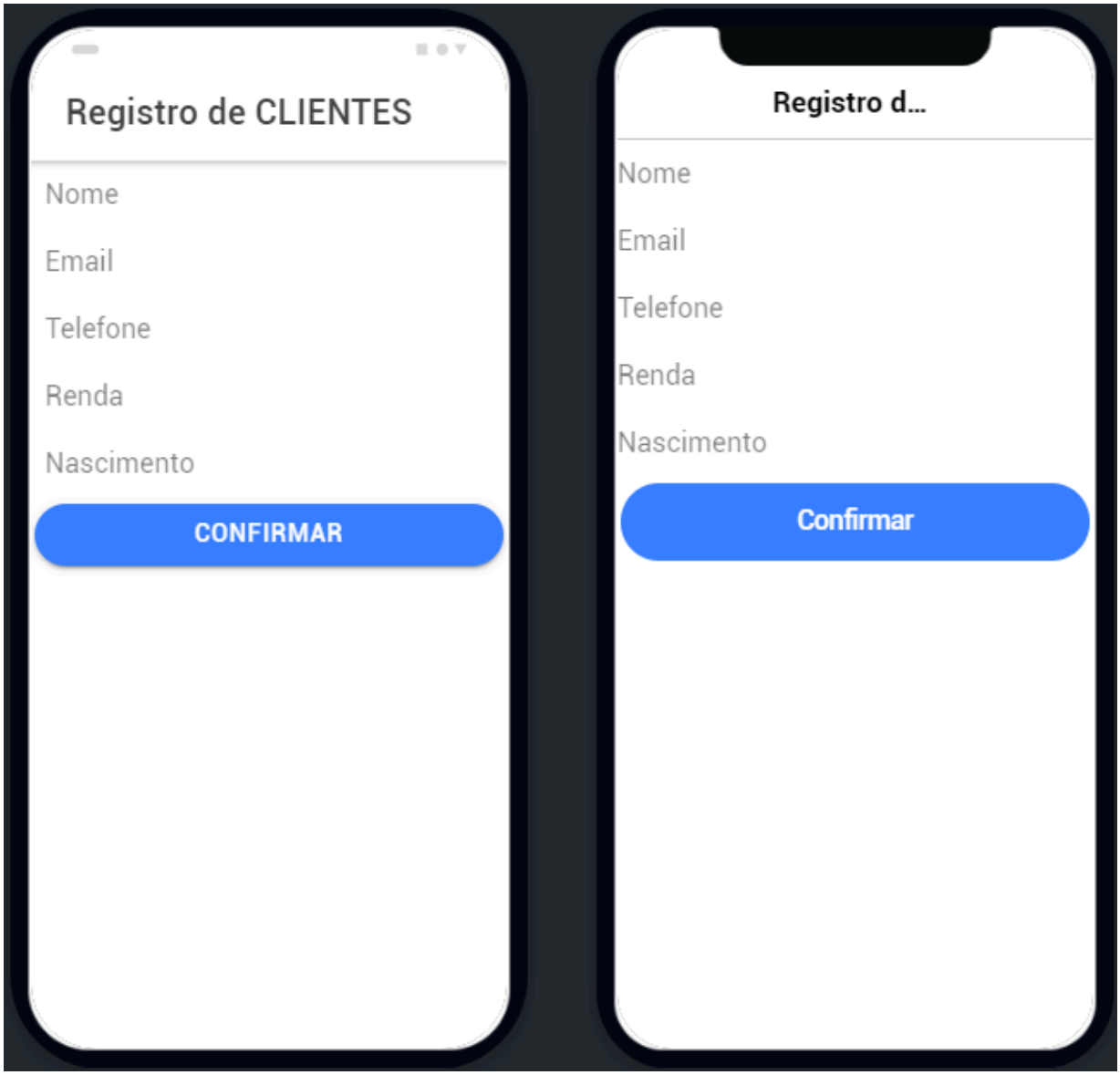


Figura 3.3: Entrada de dados de clientes.

O que acha de realizar alguns testes? Comece tirando o `placeholder` dos inputs. Depois, no botão, mude os valores para `shape`, teste novas cores, retire o `expand` e, por fim, teste sem o `padding`. O que achou?

3.5 Mudanças na interface com o usuário

A maneira como implementamos os controles de entrada na seção anterior é uma maneira de deixar a tela mais limpa, sem um excesso de controles, mas, se você começar a digitar os dados nos controles, perde-se a "dica" do que o controle representa. Uma saída para isso é utilizarmos controles de texto, os conhecidos *labels*. Vamos então adaptar nosso código para o que está representado na listagem a seguir. Vou apresentar apenas um controle, os demais você complementa, ok?

```
<ion-label position="stacked">Nome</ion-label>
<ion-input type="text" placeholder="Informe o nome"></ion-input>
```

Veja a execução do código anterior no navegador. Observou que o `<ion-label>` com o `position="stacked"` deixa o texto do label acima do input? Você pode ficar à vontade e mudar todos os outros, mas pode seguir a leitura e ver novas sugestões e, ao final, definir o que é melhor para sua aplicação. Vamos a uma nova possibilidade? Veja o código a seguir.

```
<ion-item no-padding>
    <ion-label position="floating">Nome</ion-label>
    <ion-input type="text"></ion-input>
</ion-item>
```

Veja na listagem anterior que agora estamos envolvendo os controles de interação em um componente `<ion-item>`, que normalmente serve para definição de itens comuns em uma renderização. Em nosso caso, temos a situação de que, para cada campo de entrada, precisamos de um `<ion-label>` e um `<ion-input>`. O `<ion-item>`, por padrão, apresenta uma linha na base, após a exibição dos componentes que ele mantém. Caso você queira ocultar esta linha, insira, no `<ion-item>`, o atributo `lines="none"`. O padrão é `full`.

Observe, no `<ion-item>`, o uso da diretiva `no-padding`. Se não a utilizarmos, o componente terá um espaço a mais nas quatro

bordas. Para o momento, isso não é interessante, mas recomendo testar essa situação.

Outra mudança, com um apelo visual legal, é a `position="floating"`. Ela configura o label para que ele seja exibido inicialmente na posição do input. Entretanto, quando o foco vai para o input, o label se move para cima, simulando o comportamento do `stacked`. Devido ao uso deste recurso, retiramos o `placeholder`.

Bem, e se quisermos que uma borda seja exibida nos inputs? O padrão atual normalmente utilizado em aplicações web e para dispositivos móveis não é esse, mas, para efeito de curiosidade e demonstração da facilidade de integração do Ionic com CSS, vamos trabalhar isso.

Para o teste de estilo, eu tirei o `padding` de `<ion-content>` e inseri `padding-top`, para um espaçamento apenas no topo do conteúdo. Meus componentes estão configurados como o código a seguir. A ideia de trocar o padding é pelo fato de que o `<ion-item>` também tem seu padding e, com isso, nossos componentes ficariam cada vez mais distantes da margem. Poderíamos utilizar o `no-padding` no `<ion-item>`, mas isso não retiraria o padding da margem direita.

```
<ion-content padding-top>
    <ion-item lines="none">
        <ion-label position="floating">Nome:</ion-label>
        <ion-input type="text"></ion-input>
    </ion-item>
    ...
</ion-content>
```

Agora, vamos aplicar alguns estilos para a configuração de nosso componente input para o momento em que ele receber o foco. Dessa maneira, insira os seguintes estilos no arquivo `cliente-add-edit.page.scss`.

```
ion-input.has-focus {
    background-color: rgb(250, 248, 246);
```

```
    border-radius: 10%;
    border: 1px solid black;
}
```

O estilo que estamos definindo no código anterior é para a classe CSS `has-focus`, quando estiver sendo utilizada em um componente `ion-input`. Nosso objetivo com esta configuração está em alterar a formatação do input, para ter uma cor de fundo personalizada, ter a borda arredondada, com espessura de 1px, sólida e preta. A ideia aqui é apresentar como configurarmos nossos componentes, buscando uma especialização em suas características visuais.

Uma dúvida que pode pairar é sobre como conseguimos identificar a classe CSS que devemos manipular. A resposta está na investigação dos elementos renderizados pelo navegador. O Chrome e outros navegadores trazem ferramentas que auxiliam o desenvolvedor e, como estamos utilizando o Chrome neste livro, é o que utilizarei para demonstrar o uso.

Com a aplicação em execução no navegador, clique no controle de entrada referente ao nome do cliente. Com o controle selecionado, clique com o botão direito do mouse sobre ele e escolha a opção `Inspecionar`. Você verá a janela se dividir em duas partes. Uma com sua aplicação em execução e outra com o código renderizado e em execução no navegador, tal qual é exibido na figura a seguir.

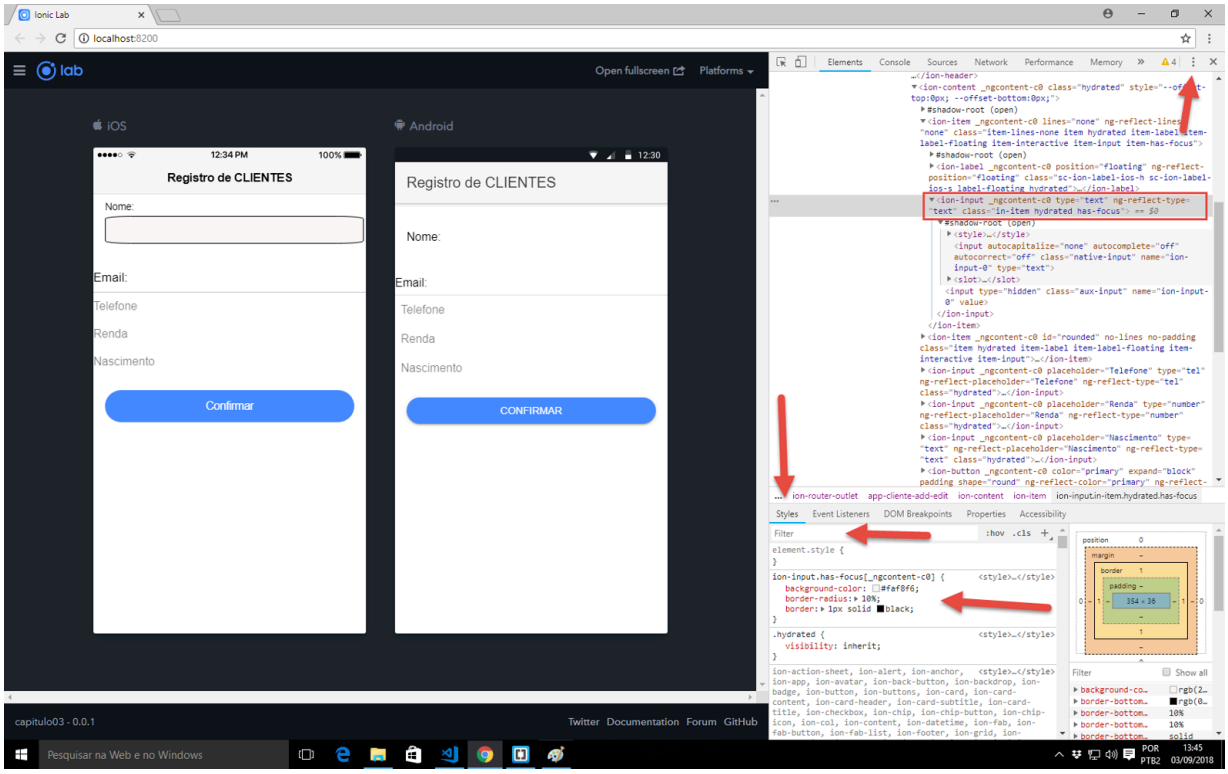


Figura 3.4: Visualização de recursos para desenvolvedor no Chrome.

Você pode alterar a disposição das informações exclusivas ao desenvolvedor clicando no botão em destaque na figura anterior e então escolhendo a posição de "dockagem" para esses recursos.

Vamos ver como identificar o código que nos interessa. Clique no controle que deseja inspecionar na aplicação e observe um efeito especial na área de código, que apontará quais trechos de código representam o elemento selecionado. É neles que devemos buscar pelas classes que queremos especializar.

Precisamos agora realizar uma formatação em relação à margem direita de nosso input, que está relacionada ao padding do `<ion-item>`. Veja o CSS na sequência, que deve ser inserido no mesmo arquivo de SCSS que inserimos no estilo anterior.

```
ion-item {
    --inner-padding-end: 3px;
```

```
    --ion-text-color: blue;
}
```

O Ionic traz uma série de variáveis pré-configuradas que ele utiliza e que podemos personalizar em nossos arquivos de estilos ligados a nossas páginas, ou, quando for o caso, no arquivo `variables.scss` dentro da pasta `theme`, na raiz da aplicação (src). Se você verificar no arquivo comentado, verá uma série de variáveis configuradas em um contexto chamado `:root {}`. As variáveis criadas ou alteradas neste contexto serão válidas para toda a aplicação. Podemos usar o `:root {}` em qualquer arquivo SCSS, mas é preciso ponderar bem esse uso, pois ele refletirá em toda a aplicação.

Voltemos a nosso estilo anterior, ligado ao componente `<ion-item>`. Estamos alterando o valor das variáveis `--inner-padding-end` para `3px` e `ion-text-color` para `blue`. Essas alterações serão válidas apenas para os componentes do tipo declarado (`<ion-item>`) no contexto da página em que estamos trabalhando. Mas como descobrir quais variáveis temos e quais devemos utilizar? O primeiro passo, com certeza, é a documentação do Ionic. Outro, mais empírico, é fazermos uso das ferramentas para o desenvolvedor do navegador e irmos alterando as variáveis disponibilizadas para vermos o comportamento delas na aplicação. Quando identificarmos o comportamento desejado, significa que encontramos a variável. A área onde devemos realizar a busca de variáveis também está em destaque na figura anterior. Veja o resultado visual das mudanças na execução de sua aplicação.

Ao alterar o `--ion-text-color`, estamos mudando a cor de todo o texto exibido em todos `<ion-item>` e em seus componentes filhos, que fizerem uso dessa variável, inclusive o que pertencer a outras tags HTML. Mas pode ser que o desejo seja mudar apenas os que pertencem diretamente ao componente. Poderíamos então utilizar o estilo a seguir. Observe que atribuímos a `color` o valor (`var`) de uma variável (`--ion-color-primary`).

```
ion-item {
    --inner-padding-end: 3px;
    color: var(--ion-color-primary);
}
```

O Ionic 6 faz uso de CSS4, o que permite o uso de variáveis para definição de estilos. Uma grande vantagem é que os valores dessas variáveis podem ser alterados em tempo de execução, diretamente no navegador.

Vamos agora alterar a cor do texto a ser digitado nos componentes `input` adicionando mais um estilo, agora para o `<ion-input>`. Veja o código na sequência e teste sua aplicação.

```
ion-input {
    color: black;
}
```

3.6 Recuperação dos dados informados

Vamos trabalhar agora a recuperação de dados informados pelo usuário. A primeira técnica que utilizaremos é relacionada a variáveis de template (*template variables*). Neste caso, atribuímos um nome a um componente, que pode ser utilizado em qualquer componente do template e também em nosso código TypeScript. Veja o código a seguir.

```
<ion-input #inputNome type="text"></ion-input>
<ion-input ref-inputEmail type="email"></ion-input>
```

Observe a inserção de `#inputNome` e `ref-inputEmail`. É dessa maneira que declaramos uma variável de template. Podemos usar o símbolo `#` antes do nome da variável, ou utilizar o prefixo `ref-`.

Mas como enviamos para a aplicação os valores da camada de apresentação? Vamos adaptar nosso `<ion-button>`, tal qual segue o

código.

```
<ion-button shape="round" color="primary" expand="block" padding
 (click)="submit(inputNome, inputEmail)">
```

Observe no código anterior o trecho `(click)="submit(inputNome, inputEmail)"`. Estamos realizando uma operação de binding de evento (*event binding*), onde associamos ao evento `click` do botão a invocação ao método `submit()`, que receberá dois argumentos, que são os objetos representados pelas variáveis. Fique atento aos parênteses que guardam o nome do evento.

Precisamos implementar esse método em nossa classe do componente, o arquivo `cliente-add-edit.page.ts`. Veja na sequência o método `submit()` implementado dentro da classe `ClienteAddEditPage`.

```
submit(inputNome, inputEmail) {
    console.log(inputNome.value + ' / ' + inputEmail.value);
}
```

Na assinatura do método anterior, não estamos tipificando os argumentos recebidos, o que, pelo padrão do TypeScript, define `any` como seus tipos. Ou seja, qualquer tipo de dado. Como sabemos que os valores que chegam no método são componentes `input`, invocamos a propriedade `value` deles, para exibir na console do navegador, os valores informados pelo usuário.

Execute sua aplicação, acesse a área do desenvolvedor pressionando `F12` ou clicando com o botão direito e então na opção `Inspecionar`. Identifique o menu `Console` na janela que se abre. Digite os valores para nome e e-mail e clique no botão. Veja que os valores digitados serão exibidos.

Poderíamos mudar a invocação do método no HTML do template, para já enviar o valor e não o componente. Poderíamos também agora tipificar nossos argumentos no método. Veja o novo código na sequência.

```
(click) = "submit(inputNome.value, inputEmail.value)"
```

```
submit(inputNome: string, inputEmail: string) {
    console.log(inputNome + ' / ' + inputEmail);
}
```

O Angular/Ionic oferece outros mecanismos, melhores e mais intuitivos para esse processo que vimos, que são conhecidos como ligação (binding). Veja o código a seguir no arquivo de TypeScript dentro da classe, antes do construtor.

```
@ViewChild('inputNome', {read: ElementRef}) nome: ElementRef;
```

Temos a definição de uma propriedade, no escopo da classe, chamada `nome`, tipificada como `ElementRef`. A propriedade está "decorada" com `@ViewChild()`. O Visual Studio Code apontará erros de compilação, pois desconhece `ViewChild` e `ElementRef`. Precisamos importar esses elementos para a classe do componente. Se você clicar nas instruções, o VSC exibirá uma lâmpada e, ao clicar nela, ele oferecerá a possibilidade de importar o que for necessário para corrigir o problema. Ao final, teremos um `import` atualizado, conforme podemos ver na sequência.

```
import { Component, OnInit, ViewChild, ElementRef } from '@angular/core';
```

Como pudemos ver, o `@ViewChild` é um decorator de propriedades. Ele realiza a busca no DOM (Document Object Model) da página pelo componente enviado como primeiro argumento. Caso você não saiba ainda o que é DOM, ele é, de maneira simplificada, uma estrutura com todos os elementos (tags) que são utilizados para a renderização de uma página (documento) HTML. Já o `ElementRef` permite o acesso direto a um componente do DOM. Em nosso exemplo, o `@ViewChild` buscará o componente e entregará uma referência para ele na propriedade `nome`, que é do tipo `ElementRef`. Neste caso, `nome` poderia ser qualquer elemento HTML.

E o `{read: ElementRef}`, o que é? Ocorre que o `@ViewChild()` buscará e retornará o elemento nomeado, que será um componente `<ion-input>` e não um componente HTML, que é o que queremos. Veja o código a seguir para o método `submit()`. Observe que invocamos

`value`, que pertence a `nativeElement`. Se não utilizarmos o `read`, não teremos essas propriedades. Recomendo que tente exibir apenas a variável `nome` na console para comprovar essa informação. Lembre-se de tirar os parâmetros na invocação de `submit()` no `<ion-button>`.

```
submit() {
    console.log(this.nome.nativeElement.value);
}
```

3.7 Uso de binding de componentes e formulários

Quando desenvolvemos uma aplicação com uma interface gráfica com o usuário, trabalhamos para que os dados informados nesta interface sejam facilmente obtidos pela classe que implementa a visão. Poderíamos pensar nessa nossa classe como a camada de controle de uma aplicação. O Ionic/Angular oferece recursos para binding (ligação) de componentes visuais com propriedades de nossa classe. Nós já vimos ligação um pouco antes, mas foi uma ligação de evento, com `(click)`. O que queremos agora é o `data binding`, uma ligação com nosso modelo de dados. Veja a implementação a seguir, em nosso HTML.

```
<ion-input [(ngModel)]="nome" type="text"></ion-input>
```

Observe o `[(ngModel)]="nome"`. Com essa codificação, estamos orientando o Ionic para que ele atribua à propriedade `nome` o valor informado no componente `<ion-input>` e, caso o valor da propriedade `nome` seja alterado no código, este valor será levado ao componente visual. Se utilizarmos apenas os `[]`, a propriedade não receberá o valor, mas o valor atribuído a ela será retornado ao componente. O que acha de testar isso? Veja a seguir o código para o método `submit()`.

```
// Nova declaração para nome
private nome: string;

submit() {
    console.log(this.nome);
    this.nome = 'Atronomogildo';
}
```

O uso de `()` no `<ion-input>` garante que o valor seja alterado na propriedade, pois ele invoca o evento `change` do componente. Tente implementar a seguinte instrução, mantendo o mesmo código para `submit()`. Teste e verifique que agora o modelo é atualizado com o dado do componente, mas a instrução anterior é mais curta e resolve a situação.

```
<ion-input [ngModel]="nome" (ngModelChange)="nome = $event" type="text">
</ion-input>
```

Voltando à premissa de que uma aplicação Ionic segue o modelo de aplicações web, precisamos reforçar que o HTML faz uso de formulários para os controles de entrada de dados por parte do usuário e consequente submissão destes dados para uma aplicação servidora. No Ionic, esta aplicação servidora por enquanto é nossa classe do componente. Vamos adaptar nosso código do template `cliente-add-edit.page.html` para o que apresento na sequência.

```
<ion-content padding-top>
  <form (ngSubmit) = "submit()">
    <ion-item lines="none">
      <ion-label position="floating">Nome</ion-label>
      <ion-input [(ngModel)]="nome" name="nome" type="text"></ion-input>
    </ion-item>
    <!-- demais itens -->
    <ion-button shape="round" color="primary" expand="block" padding
type="submit">
      Confirmar
    </ion-button>
  </form>
</ion-content>
```

Observe a definição de um elemento `<form>` com um *event binding* para `ngSubmit`, que invocará o método `submit()` quando o `<ion-button>` for pressionado. Veja, no botão, a definição de `type="submit"`, o que torna o componente o responsável pelo envio dos dados englobados pelo formulário para o método definido em `form`. Peço que veja também que ao `<ion-input>` adicionamos agora um novo atributo, o `name`, que é uma exigência do Ionic/Angular quando temos componentes de entrada de dados dentro de um `<form>`. Isso possibilitará uma varredura, pelo framework, no DOM dos componentes filhos do formulário.

Vimos anteriormente que podemos criar propriedades em nossa classe que recebam e forneçam os dados de e para nossos componentes visuais. Seguindo o modelo de implementação que estamos trabalhando, teríamos uma propriedade para cada `<ion-input>`. Entretanto, se pensarmos em um contexto orientado a objetos, todos os dados solicitados pelo formulário pertencem a um conjunto comum, que são os dados de um cliente, o que nos leva a uma classe.

No momento, não criaremos classes para nosso modelo de negócio `Cliente`. Vamos fazer uso de um recurso do JavaScript, de criar um objeto que terá suas propriedades baseadas nos dados vindos do template. Veja na sequência a declaração da propriedade `cliente` como um objeto JavaScript vazio que deve ser feita antes do construtor, em nossa classe.

```
cliente = {};
```

Na sequência, precisamos ajustar nosso template, que precisará fazer uso dessa nossa propriedade. Veja o código para `nome`, na sequência. Veja que o valor para o *data binding* agora é composto pelo nome de nossa propriedade e da propriedade que queremos utilizar para o nosso objeto `cliente`.

```
<ion-input [(ngModel)]="cliente.nome" name="nome" type="text"></ion-input>
```

Depois de alterar todos os `<ion-input>`, inserindo uma propriedade para `cliente`, de acordo ao solicitado no componente, altere o método `submit()` para o código a seguir.

```
submit() {
    console.log(this.cliente);
}
```

Informe os dados nos componentes, clique/pressione o botão de confirmação e veja o resultado na console do navegador, tal qual mostra a figura a seguir. Veja os nomes das propriedades para `cliente`.

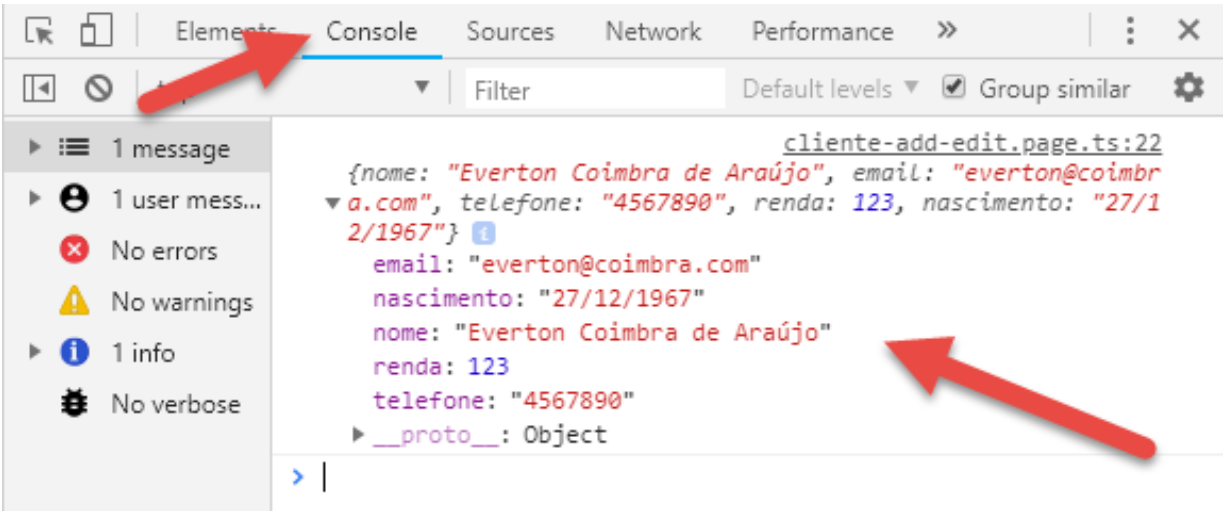


Figura 3.5: Estrutura criada para o projeto.

A técnica que utilizamos na seção que estamos terminando agora é conhecida como `Template Driven Forms`, onde temos nosso formulário guiado por um modelo, nosso template. Vimos que nos controles indicamos o *data binding* por meio do `ngModel` e que recuperamos em nossa classe os dados informados no template, mas, que também podemos popular nossos componentes com valores atribuídos em nossa propriedade de classe.

3.8 Angular Reactive Forms

De acordo com o site do Angular, o Reactive Forms fornece uma estratégia guiada ao modelo para gerenciar entrada de dados em um formulário onde esses valores sofrem alteração, quer seja de maneira interna, ou por interação com o usuário. Vamos adaptar nosso componente de clientes para utilizar este recurso.

Nosso primeiro passo é compreender como podemos adicionar o Reactive Forms à nossa aplicação. É preciso entender que o Ionic/Angular trabalha com o conceito de módulos, e, nesses módulos, temos componentes. Quando criamos nossa página de clientes, usamos o Ionic CLI para gerar um componente, que é do tipo `page`. Veja a pasta do componente e verifique a existência de um arquivo chamado `cliente-add-edit-module.ts`. Com este arquivo aberto, observe a declaração da propridade `declarations` no objeto enviado ao decorator `@NgModule()`. Esta instrução informa ao Ionic que o módulo `ClienteAddEditPageModule`, que é uma classe, declara que tem implementado nele a classe `ClienteAddEditPage`, que, se você observar, está declarada nos imports, trazendo para a referência dela o artefato `cliente-add-edit.page`.

Temos então um componente que é composto por uma classe TypeScript e um template HTML, que pertence a um módulo. Nossa situação agora é a seguinte: precisamos usar o `Angular Reactive Forms`, que é um componente, mas, para termos acesso a um componente, precisamos ter acesso ao módulo a que ele pertence. Desta maneira, voltando ao nosso módulo de clientes, observe a propriedade `imports` no decorator `@NgModule()`. Temos importados vários componentes. Precisamos informar que necessitamos importar para nosso módulo o módulo responsável pelo `Reactive Forms`. Sendo assim, adicione a seguinte instrução ao final do `imports`. Fique atento à importação que será necessária.

```
ReactiveFormsModule
```

Perfeito, já podemos trabalhar no uso do `Reactive Forms` em nosso componente. Abra o arquivo `cliente-add-edit-page.ts`. Precisamos declarar uma propriedade que gerenciará nosso formulário. Também precisaremos ter em nossa classe um objeto `FormBuilder` e receberemos este objeto pelo construtor. Veja o código a seguir. Fique atento (a) à necessidade de importações para as novas classes utilizadas.

```
clienteForm: FormGroup;

constructor(private formBuilder: FormBuilder) {
}
```

A primeira explicação que veremos é o porquê de `private` antes do argumento. A resposta é simples e está relacionada a escopos. Se não utilizarmos um modificador de escopo, o argumento terá visibilidade apenas dentro do construtor. Ao utilizarmos `private`, estamos informando que o argumento será visto como um campo privado para o componente, ou seja, visível na classe. Poderíamos usar `public`, o que transformaria o argumento em propriedade, visível por classes consumidoras e também pelo nosso template.

Quando temos um formulário HTML, temos dentro dele o que chamamos de controles filhos, pertencentes ao formulário e que normalmente possuem os valores informados pelo usuário. Estes controles filhos, no `Reactive Forms`, são chamados de `FormControl`. Um objeto `FormGroup` agrega nele os valores de todos os `FormControls` pertencentes ao formulário em questão.

O `FormBuilder` é uma classe que possibilita a criação de diversos componentes relacionados ao `Reactive Forms`, dentre eles, o `FormGroup`. Normalmente, quando precisamos de algum serviço oferecido pelo framework, nós não vamos buscar por ele, apenas informamos que precisamos dele e que ele deve ser fornecido para o componente que informa essa necessidade. Isso é conhecido como injeção de dependência e essa injeção é realizada pelo construtor. Desta maneira, quando declaramos, no código anterior, que

receberemos um argumento chamado `formBuilder`, ao tipificarmos, o framework já o fornecerá para nosso componente e o disporá como uma propriedade privada de nossa classe.

Com a declaração de nosso `FormGroup` e `FormBuilder` realizada, precisamos agora atribuir um objeto de grupo à nossa propriedade `clienteForm`. Se você conhece Orientação a Objetos, talvez fique tentado a realizar essa configuração no construtor, mas o padrão no Ionic/Angular (e também recomendado para outras linguagens) é realizarmos isso em um método de inicialização. Se você olhar atentamente para a declaração de nossa classe, verá que ela implementa a interface `OnInit` e que, quando geramos a página pelo Ionic CLI, um método chamado `ngOnInit()`, desta interface, está declarado, mas sem comportamento. É nele que configuraremos nosso formulário. Veja o código a seguir. Pode ser necessário importar `Validators`.

```
ngOnInit() {
    this.clienteForm = this.formBuilder.group({
      nome: ['', Validators.compose([Validators.required,
Validators.minLength(3), Validators.maxLength(50)])],
      email: ['', Validators.compose([Validators.required,
Validators.email])],
      telefone: ['', Validators.required],
      renda: ['0', Validators.compose([Validators.required,
Validators.min(0)])],
      nascimento: ['', Validators.required]
    });
}
```

Veja a invocação ao método `group()`, enviando a ele um objeto JavaScript, com propriedades nomeadas de acordo com o que teremos definido em nosso template e adicionando ainda validadores padrões para essas propriedades. Em algumas, temos apenas uma validação a ser realizada, em outras, temos mais de uma, e fazemos uso do método `compose()` para isso. Veja que, para `renda`, estamos declarando um valor inicial, `0`, para ser exibido no controle. O `Reactive Forms` traz um conjunto de validadores já implementados e

à disposição, mas também nos dá a possibilidade de criação de validadores personalizados.

Como mudamos a maneira de ligação de nossos controles de interação com nosso componente, precisamos alterar nossos controles no template. Veja o código a seguir.

```
  <form [formGroup]="clienteForm" (submit)="submit()">
    <ion-item lines="none">
      <ion-label position="floating">Nome</ion-label>
      <ion-input formControlName="nome" type="text"></ion-input>
    </ion-item>
     <ion-item lines="none">
        <ion-label position="floating">Email</ion-label>
        <ion-input formControlName="email" type="email"></ion-input>
    </ion-item>
    <ion-item lines="none">
      <ion-label position="floating">Telefone</ion-label>
      <ion-input formControlName="telefone" type="tel"></ion-input>
    </ion-item>
    <ion-item lines="none">
      <ion-label position="floating">Renda</ion-label>
      <ion-input formControlName="renda" type="number"></ion-input>
    </ion-item>
    <ion-item lines="none">
      <ion-label position="floating">Nascimento</ion-label>
      <ion-input formControlName="nascimento" type="text"></ion-input>
    </ion-item>
    <ion-button shape="round" color="primary" expand="block" padding
type="submit">
      Confirmar
    </ion-button>
  </form>
```

Temos um *data binding* para `formGroup`, que recebe nossa variável definida no componente, e a mudança para o *event binding*, que agora é o `submit`. Esta é a sintaxe para um formulário utilizando o `Reactive Forms`. Outra mudança está nos `<ion-input>`, que não têm mais binding com um objeto do componente e tampouco a definição

de `name`. No lugar disso, temos a definição de `formControlName`, recebendo o nome da propriedade enviada na criação de nosso `formGroup` para a variável `clienteForm`. Estes nomes devem ser IDÊNTICOS.

Nós temos toda a burocracia criada para binding do formulário com nosso componente e também temos as validações, mas, se clicarmos/pressionarmos o botão, nada ocorre. Você pode pensar: vamos aplicar as validações no momento em que os dados forem submetidos. E você não está errado, esta é sim uma possibilidade. Mas vamos utilizar um recurso de validação "in-line" dos controles. Insira o código a seguir abaixo dos respectivos `<ion-input>`.

```
<!-- NOME -->
<small class="error-message"
    *ngIf="clienteForm.get('nome').errors?.required ||
clienteForm.get('nome').errors?.minlength ||
clienteForm.get('nome').errors?.maxlength">
    Informe o nome (3 a 50 caracteres)
</small>

<!-- EMAIL -->
<small class="error-message"
    *ngIf="clienteForm.get('email').errors?.required ||
clienteForm.get('email').errors?.email">
    Informe um email válido
</small>

<!-- TELEFONE -->
<small class="error-message"
    *ngIf="clienteForm.get('telefone').errors?.required">
    Informe o telefone
</small>

<!-- RENDA -->
<small class="error-message"
    *ngIf="clienteForm.get('renda').errors?.required ||
clienteForm.get('renda').errors?.min ">
    Informe uma renda positiva
```

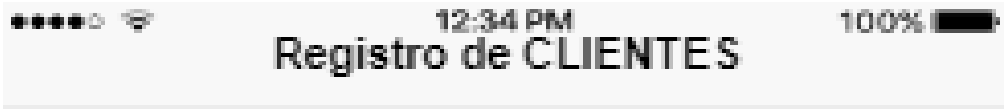
```
</small>

<!-- NASCIMENTO -->
<small class="error-message"
    *ngIf="clienteForm.get('nascimento').errors?.required">
    Informe uma data de nascimento
</small>
```

Note que ao utilizarmos a tag HTML `<small>`, estamos utilizando uma classe de estilo. Precisamos criá-la em nosso arquivo de CSS, tal qual podemos ver na sequência.

```
small.error-message {
    color: var(--ion-color-danger);
}
```

Ainda, em relação à tag `<small>`, do código anterior, observe que utilizamos uma diretiva do Angular, chamada `*ngIf`. Esta diretiva recebe o retorno de uma expressão lógica e, se este retorno for verdadeiro, o componente que a declara é renderizado. Você pode verificar que nossa expressão está relacionada à identificação de existência de erros relacionados às validações registradas para cada propriedade do formulário. Se o valor for verdadeiro, então, como dito, o texto de alerta é exibido. Veja a figura a seguir, que apresenta nosso template em execução e com as validações aplicadas.



Nome

Informe o nome (3 a 50 caracteres)

Email

Informe um email válido)

Telefone

Informe o endereço

Renda

-1

Informe uma renda positiva

Nascimento

Informe uma data de nascimento

Confirmar

Figura 3.6: Validação para os controles de entrada do template.

Para finalizar a explicação dessa técnica de validação, podemos implementar um código que habilite o botão de confirmação dos dados apenas quando o formulário não possuir mais erros. Para isso, veja o código para o botão na sequência. Identifique nele o binding com `disabled` e a validação do formulário.

```
<ion-button shape="round" color="primary" expand="block" padding
type="submit" [disabled]="!clienteForm.valid">
      Confirmar
</ion-button>
```

3.9 Não exibir mensagens de erro antes da submissão e alteração

Com a técnica que trabalhamos na seção anterior, assim que acessamos nossa interface com o usuário já nos deparamos com as mensagens de erro, pois, a princípio, os controles não estão preenchidos e a mensagem é então exibida. Mas e se quisermos que essa mensagem seja exibida apenas após termos tentado submeter os dados informados?

Para implementarmos a exibição de mensagens de validação apenas após a tentativa de submissão, precisamos inicialmente tirar do botão de confirmação a validação que temos para habilitação/desabilitação dele, pois o usuário precisará submeter os dados para que eles possam ser verificados e, em caso de requisitos não cumpridos, para que as mensagens sejam então exibidas. Desta maneira, vamos deixar nosso componente de botão tal qual o código a seguir.

```
<ion-button shape="round" color="primary" expand="block" padding
type="submit">
      Confirmar
</ion-button>
```

Na sequência, precisamos colocar a validação retirada do botão no método que captura a submissão de nossos dados. Veja essa implementação na sequência. Verifique nela que, caso haja algum problema com os dados do formulário da página, a execução do método é interrompida. Poderíamos, desta maneira, implementar o processamento dos dados após o `if()`, o que garante que ele ocorrerá apenas quando os dados forem válidos.

```
submit() {
    if (this.clienteForm.invalid || this.clienteForm.pending) {
      return;
    }
}
```

Com a implementação que fizemos, o problema ainda não está resolvido, pois as mensagens de erro continuam sendo exibidas logo que a página é exibida. Precisamos de uma variável de controle, que identificará quando nosso formulário já teve uma submissão realizada. Vamos então fazer uso de uma variável de template em nosso componente de formulário. Veja o código a seguir.

```
<form #form="ngForm" [formGroup]="clienteForm" (submit)="submit()">
```

Observe a declaração `#form="ngForm"`. Com esta instrução, estamos atribuindo a `form` o objeto que representa o formulário, via Angular, por meio do `ngForm`. Com isso, temos acesso a informações do formulário e poderemos utilizá-las para a exibição ou não das mensagens de erro apenas após a tentativa de submissão. Veja na sequência a adaptação para o `*ngIf` do componente de nome.

```
<small class="error-message"
        *ngIf="(clienteForm.get('nome').errors?.required ||
clienteForm.get('nome').errors?.minlength ||
clienteForm.get('nome').errors?.maxlength) &&  form.submitted">
    Informe o nome (3 a 50 caracteres)
</small>
```

Englobamos as condicionais de requisitos de validação entre parênteses e adicionamos um condicional `&&` para `form.submitted`. A

mensagem só será exibida se os dados estiverem inválidos e se o formulário já tiver sido submetido. O que acha de adaptar os demais controles e testar a aplicação?

Seguindo essa linha de exibição de mensagem apenas após algum tipo de interação, como fizemos para a submissão, podemos ter a possibilidade de exibição da mensagem após o usuário sair do controle visual. Isso é também possível e bem simples. Agora, para variarmos, vamos ver esta alteração no componente para telefone. Veja o código a seguir.

```
<small class="error-message"
      *ngIf="clienteForm.get('telefone').errors?.required &&
(form.submitted || clienteForm.get('telefone').touched)">
    Informe o telefone
</small>
```

Verifique que inserimos parênteses para o `form.submitted` e adicionamos um `OU` para verificar se a propriedade do formulário, chamada `telefone`, foi `tocada`, ou seja, passou por um processo de validação. Vamos testar nos demais controles?

Vamos aproveitar e ver um novo recurso visual para apresentar controles com regras não cumpridas em relação às validações. Veja a seguir a nova escrita para o `<ion-input>`. Observe o binding para uma classe CSS chamada `invalid` e note as regras que farão com que ela seja utilizada.

```
<ion-input formControlName="nome" type="text"
[class.invalid]="!clienteForm.controls['nome'].valid &&
clienteForm.controls['nome'].touched"></ion-input>
```

Precisamos, para utilizar a classe de estilo `invalid`, implementá-la no arquivo de estilos de nossa página. Veja a seguir o código para ela, que podemos inserir ao final de nosso arquivo. Apenas mais uma dica de validação e integração de nossa página com CSS.

```
.invalid {
    border-bottom: 1px solid #ff6153;
```

```
}
```

3.10 Validação do Reactive Form no componente TypeScript

Vimos na seção anterior que uma das possibilidades de validação de um formulário é exibir, próximo a seu componente de entrada, uma mensagem de aviso ao usuário sobre as validações não cumpridas. Durante os testes, você pode ver que conforme atendem às validações, as mensagens desaparecem, pois estão condicionadas à existência do erro. Trabalhamos também a situação de habilitar o botão de confirmação apenas para quando todas as validações tiverem sido atendidas.

Mas, e se a necessidade, ou desejo, for de exibir os erros apenas após o usuário clicar/pressionar o botão de confirmação? Bem, se você notou a condição de não habilitação do botão, ela faz uso de uma propriedade do formulário que nos informa esta situação e podemos inicialmente trabalhar com ela.

Para testarmos isso, vamos comentar nossos elementos `<small>` e, em nosso componente, vamos criar uma propriedade, que indicará a existência de erros durante o processo de validação e outra para receber a mensagem de erro. A propriedade `hasErrors` será utilizada como condição para exibirmos, ou não, mensagens de erro de validação. Precisamos também deixar habilitado nosso `<ion-button>`, desta maneira, retire o *data binding* que controla o estado dele, de acordo com as validações do formulário. Veja a implementação na sequência, que é seguida de uma implementação a ser realizada em nosso template, logo após a definição de nosso `<form>`.

```
// arquivo TS, antes do construtor
hasErrors = false;
errorMessage: string;
```

```
<!-- arquivo HTML, após <form> -->
<ion-label *ngIf="hasErrors">
  Erro:
    {{ errorMessage }}
</ion-label>
```

Notou que inicializamos a propriedade `hasErrors` com `false`? Isso impedirá, de imediato, que o `<ion-label>` de erros seja exibido. Já dentro do componente, temos uma `Angular Expression`, que contém a propriedade a ser invocada e ter seu resultado renderizado. A sintaxe é colocar a expressão entre chaves duplas, `{{}}`.

Vamos agora implementar uma técnica de validação no método que será invocado na submissão do formulário, o nosso já conhecido `submit()`. Veja o código para ele na sequência.

```
submit() {
    if (this.clienteForm.invalid) {
      this.hasErrors = true;
      this.errorMessage = 'Existem erros nos dados informados';
    } else {
      this.hasErrors = false;
    }
}
```

Verificamos se o formulário `clientesForm` é inválido e, caso seja, atribuímos `true` a `hasErrors` e uma mensagem a `errorMessage`. Caso não existam erros no formulário, por segurança, a variável de controle recebe `false`. Após essa validação, poderemos delegar a continuidade do fluxo do processo. Teste sua aplicação e, sem informar nada, clique/pressione o botão de confirmação.

Mas não tem como realizar as validações para cada controle de entrada que configuramos para o formulário? É claro que tem! Vamos a algumas alterações para isso. A primeira será alterar o tipo de dado de nossa `errorMessage` para uma matriz de `string`. Veja o código a seguir e observe que aproveitei para pluralizar o nome da propriedade.

```
errorsMessage: string[];
```

Com esta alteração, vislumbramos um elemento da matriz para cada erro que possa existir durante o processo de validação. Vamos então à nova implementação para o método `submit()`. Veja-a na sequência.

```
submit() {
    this.errorsMessage = [];
    if (this.clienteForm.get('nome').hasError('required')) {
      this.errorsMessage.push('Nome é obrigatório');
    }
    if (this.clienteForm.get('email').hasError('required')) {
      this.errorsMessage.push('Email é obrigatório');
    }
    this.hasErrors = this.errorsMessage.length > 0;
}
```

Observe, após a inicialização da propriedade `errorsMessage`, uma condição que verifica a existência de um erro para a validação `required` para uma propriedade de `clienteForm`, chamada `nome`. Em caso positivo, adicionamos à matriz de mensagens de erro uma string apontando isso. Fazemos o mesmo para o `email`. Ao final, temos a propriedade `hasErrors` recebendo um valor booleano, de acordo com a existência ou não de elementos na matriz de erros. Se um erro for adicionado, ele deve ser exibido para o usuário.

Mas como exibir agora os vários possíveis erros? Poderíamos apenas ajustar o nome na Angular Expression para a nova propriedade. Entretanto, as mensagens apareceriam uma do lado da outra, e nosso objetivo é que elas apareçam uma embaixo da outra.

O que precisamos é de uma maneira de exibir o conteúdo da propriedade como uma listagem, linha a linha. O Angular/Ionic nos oferece uma diretiva, `*NgFor`, que nos possibilita uma iteração em uma coleção de dados, isto é, poderíamos ter um `<ion-label>` para cada elemento da propriedade. Nosso código poderia ser o que está

na sequência, em substituição ao que já temos para a exibição do erro.

```
<ion-label *ngFor="let errorMessage of errorsMessage">
  {{errorMessage}}<br/>
</ion-label>
```

Esta diretiva executa uma varredura, elemento a elemento de uma coleção, em nosso caso, `errorsMessage` e, para cada elemento, a cada iteração, ocorre seu armazenamento, por meio do `let`, em uma variável chamada de `errorMessage`. Desta maneira, podemos ter a Angular Expression apontada no código anterior, que será exibida quantas vezes for a quantidade de elementos na coleção. O que acha de testar a aplicação?

Você deve ter percebido pelo código que, ao termos a página renderizada, antes de confirmarmos os dados, um `<ion-label>` foi renderizado. Isso é algo esperado, pois não temos uma condição para exibição ou não dos erros. Se você confirmar, os erros serão exibidos, um em cada linha. Mas não queremos perder o espaço ocupado por elemento inexistente quando a página for renderizada. O que podemos fazer?

Para evitarmos o problema do parágrafo anterior, você talvez tenha pensado na `*NgIf`, que já utilizamos. Você está correto (a), mas não é possível termos duas diretivas em um mesmo componente Ionic. Como queremos uma listagem onde cada `<ion-label>` esteja em uma linha, podemos utilizar o componente `<ion-list>` e nele realizar a verificação de exibição ou não e, dentro dele, ter o `<ion-label>`. Veja nosso código na sequência. Teste sua aplicação após a implementação e veja o resultado antes e após a confirmação dos dados.

```
<ion-list *ngIf="hasErrors">
    <ion-label *ngFor="let errorMessage of errorsMessage">
      {{errorMessage}}<br/>
    </ion-label>
</ion-list>
```

O que acha de estilizarmos essa exibição de erros? Na sequência, uma sugestão apenas para você verificar a possibilidade. Lembre-se de que este código deve estar em nosso arquivo de estilos.

```
ion-list.errors-summary {
    border: 1px solid black;
    background-color: red;
    margin-bottom: 0px;
}

ion-label.errors-summary {
    color: white;
    text-align: center;
    margin-bottom: 0px;
    margin-top: 0px;
}
```

Após a implementação dos estilos, precisamos utilizá-los, sendo assim, insira o uso das classes nos componentes com a instrução `class="errors-summary"` dentro das tags iniciais deles (`<ion-list>` e `<ion-label>` . Teste novamente sua aplicação e veja o resultado.

Podemos aplicar outra técnica para a exibição dos erros de validação sem a necessidade de fazermos verificações na submissão do formulário. Ela pode ser um pouco mais burocrática, mas pode ser interessante conhecermos. Comente em seu código, tanto do template HTML, como no componente TypeScript, nossos tratamentos de erro para testarmos essa nova técnica.

Para facilitar, comentei `hasErrors = false;` e `errorsMessage: string[];` na classe. Consequentemente, comentei o código do método `submit()` e o `<ion-list>` da mensagem de erro após o `<form>` .

Vamos começar definindo uma propriedade de um objeto JavaScript. Este objeto terá propriedades de acordo com o nome das propriedades de nosso formulário. Cada propriedade dessa terá como valor uma matriz de objetos, composta pelo tipo de validação que cada componente do formulário tem, e de uma mensagem de

erro a ser aplicada quando a validação for malsucedida. Veja o código na sequência. Implementei apenas para `nome` e `renda`. Você pode, se quiser, implementar para as outras propriedades do formulário, que deve estar antes do construtor, junto com as demais propriedades.

```
validationMessages = {
    nome: [
        { type: 'required', message: 'Nome é obrigatório' },
        { type: 'minlength', message: 'Nome deve ter ao menos 3
caracteres' },
        { type: 'maxlength', message: 'Nome não pode ter mais que 50
caracteres' }
      ],
    renda: [
        { type: 'min', message: 'Renda precisa ser positiva' }
      ]
};
```

Agora que temos as validações mapeadas para um objeto específico de mensagens de erro, precisamos ajustar nosso template para fazer uso deste objeto e de nossas validações. Sendo assim, insira, logo depois da abertura de `<form>`, o código a seguir.

```
<div>
  <ng-container *ngFor="let validation of validationMessages.nome" >
    <div *ngIf="clienteForm.get('nome').hasError(validation.type)">
      {{ validation.message }}
    </div>
  </ng-container>
  <ng-container *ngFor="let validation of validationMessages.renda" >
      <div *ngIf="clienteForm.get('renda').hasError(validation.type)">
        {{ validation.message }}
      </div>
  </ng-container>
</div>
```

Consegue abstrair o código anterior? No exemplo, deixei de utilizar componentes Ionic, pois essa é uma possibilidade que podemos ter

também, utilizarmos HTML e componentes Angular. Usamos as diretivas `*NgFor` e `*NgIf`, que já conhecemos. Execute sua aplicação e teste. Veja que os erros que existem, já aparecem no topo da página e, conforme formos informando os valores corretos, os erros deixam de ser exibidos.

3.11 Um componente específico para data

Como você certamente realizou diversos testes em nossa aplicação, já notou que a informação da data de nascimento do cliente ocorre em um controle de entrada normal para texto. Teríamos que informar a data em um formato e valor corretos. Mas, usando algumas aplicações em nossos dispositivos, vemos a existência de um controle especial para informação de uma data. O Ionic nos oferece isso também. O componente que utilizaremos é o `<ion-datetime>`, que tem vários exemplos em sua documentação, mas no código a seguir, trago uma implementação mais elaborada, fazendo uso de `popup` para exibir o calendário. Dê uma lida com calma.

```
<ion-grid class="ion-no-padding">
      <ion-row>
        <ion-col size="9">
          <ion-item lines="none">
            <ion-label position="floating">Nascimento</ion-label>
            <ion-input [value]="dataSelecionada" [readonly]="true"></ion-
input>
          </ion-item>
        </ion-col>
        <ion-col class="ion-no-padding">
          <ion-item lines="none">
            <ion-button fill="clear" id="open-date-input" class="ion-no-
margin;ion-no-padding">
              <ion-icon icon="calendar"></ion-icon>
            </ion-button>
            <ion-popover trigger="open-date-input" show-backdrop="false"
```

```
[dismissOnSelect]="true">
              <ng-template>
                <ion-datetime #popoverDatetime presentation="date"
formControlName="nascimento"
                  displayFormat="DD/MM/YYYY" min="1967" max="2022-12-31"
                  (ionChange)="dataSelecionada = confirmarData()">
                </ion-datetime>
              </ng-template>
            </ion-popover>
          </ion-item>
        </ion-col>
      </ion-row>
</ion-grid>
```

Vamos agora executar nossa aplicação. Observe que o componente é renderizado quando interagimos com o botão que tem um calendário como ícone, da mesma maneira no Android e iOS. Mudam alguns detalhes visuais, que são padrões para cada plataforma, mas não é feito uso dos recursos nativos delas, não apresentam o visual específico para este controle. Precisamos trazer para nossa aplicação componentes que possibilitem esse acesso nativo.

Felizmente, podemos fazer uso de plugins e existem vários, que você pode verificar em https://ionicframework.com/docs/native/. Nós faremos uso agora do `Date Picker`. Veja a seguir as instruções que precisam ser executadas no terminal para a instalação deste plugin. Observe que, mesmo utilizando o Capacitor, podemos ter plugins Cordova e, neste caso, este plugin está na documentação atual do Ionic.

```
// Instalação do plugin
npm install cordova-plugin-datepicker

// Instalação do acesso nativo ao plugin
npm install @ionic-native/date-picker

// Instalação da dependência para uso do plugin
npm install @ionic-native/core
```

Após a instalação ter ocorrido com sucesso, precisamos informar à nossa aplicação que faremos uso deste plugin. Um plugin também é visto como um módulo para as aplicações Ionic. Um detalhe importante para o caso do `DatePicker` é que ele será visto como um serviço/recurso, que poderá ser injetado em outros componentes. Desta maneira, precisamos registrá-lo no arquivo `app.module.ts` na propriedade `providers`, mas, antes disso, precisamos importá-lo para nosso módulo principal, o `app.module.ts`. Veja a instrução a seguir a ser inserida no início do arquivo.

```
import { DatePicker } from '@ionic-native/date-picker/ngx';
```

Agora sim, vamos então inserir `DatePicker` no elemento `providers`. Para auxiliar, na sequência, veja este elemento com a inserção.

```
providers: [{ provide: RouteReuseStrategy, useClass: IonicRouteStrategy,
}, DatePicker],
```

Precisamos agora realizar a implementação deste plugin em nosso componente/página de clientes. Não temos um componente Ionic (tag) para inserirmos este plugin em nosso template, o que nos levará a utilizar o código apresentado na sequência, mas você pode depois ajustar o layout à sua preferência. Verifique que no `<div>`, temos um evento `click`, mapeado para um método `selecionarData()`, que será responsável por renderizar o `DatePicker`.

```
<div (click)="selecionarData()" class="ion-no-padding">
      <ion-grid>
        <ion-row>
          <ion-col size="9" class="ion-no-padding">
            <ion-item lines="none">
              <ion-label position="stacked">Nascimento</ion-label>
              <ion-input formControlName="nascimento"></ion-input>
            </ion-item>
          </ion-col>
          <ion-col class="ion-no-padding">
            <ion-item lines="none">
              <ion-button fill="clear" class="ion-no-margin;ion-no-
padding">
```

```
              <ion-icon icon="calendar"></ion-icon>
            </ion-button>

          </ion-item>
        </ion-col>
      </ion-row>
    </ion-grid>
</div>
```

Agora, resta-nos implementar este método em nossa classe responsável pelo template. Porém, antes disso, precisamos ter injetado em nossa classe o `DatePicker` e, para isso, precisamos inserir a seguinte declaração em nosso construtor. Veja o código na sequência.

```
private datePicker: DatePicker
```

Com o objeto/serviço/componente disponibilizado para nossa classe, vamos então implementar o método `selecionarData()`:

```
selecionarData() {
    this.datePicker.show({
      date: new Date(),
      mode: 'date',
      locale: 'pt-br',
      doneButtonLabel: 'Confirmar',
      cancelButtonLabel: 'Cancelar'
    })
    .then(
      data =>
this.clienteForm.controls.nascimento.setValue(data.toLocaleDateString())
    );
}
```

Verifique que invocamos o método `show()` enviando a ele um objeto JavaScript com configurações que personalizarão a renderização do componente. Esse método retorna um `Promise<Date>` onde `Date` é utilizado como tipo genérico para `Promise`. Mas o que é uma `Promise`?

Quando executamos uma tarefa de maneira síncrona, aguardamos ela terminar seu processo antes da execução do código passar para uma seguinte instrução. Quando precisamos que as tarefas sejam executadas, sem dar a impressão de congelamento da aplicação, precisamos fazer uso de programação e execução assíncrona.

O ECMAScript 6 (ES6) trouxe uma API conhecida como `Promise`, que possibilita a resolução dessas chamadas. Podemos então dizer que uma `Promise` é um retorno para uma chamada de uma função assíncrona que, ao ser concluída, devolverá ao chamador o fluxo da execução da aplicação e então, com o retorno à chamada, por meio de funções de callback, executamos funcionalidades que deveriam ser executadas quando a `Promise` for concluída.

A arrow function que temos em `then()` será executada apenas quando o método `show()` for totalmente concluído. Em nosso caso, é a renderização do componente para seleção de uma data. Veja que `then()` recebe um valor para a variável `data`, que será a data selecionada. Esta data então é formatada por `toLocaleDateString()` e atribuída para nosso componente de formulário relacionado à data de nascimento, que é nosso `<ion-datetime>`, que renderizará a data escolhida.

Retornemos ao objeto enviado para configuração. A propriedade `date` receberá a data atual como data inicial a ser exibida; `mode` recebe o que deverá ser renderizado pelo controle, em nosso caso, datas. `locale` trabalha a localização a ser utilizada pelo controle, em nosso caso, Português/Brasil; enfim, `doneButtonLabel` e `cancelButtonLabel` são nomes semanticamente compreendidos.

Precisamos realizar algumas alterações no template. Troque o `position` do `<ion-label>` para `stacked`, pois não teremos o `<ion-datetime>` com interação, então melhor deixarmos o label já acima da data. O que acha agora de executarmos a aplicação para testarmos?

Ao executarmos a aplicação, vemos que o `<ion-input>` disponibiliza o conteúdo para a interação, mas, como ele está dentro do `<div>`

clicável, o picker é exibido. Vamos então desabilitar a interação diretamente em nosso formulário, ajuste a declaração como o código a seguir apresenta.

```
nascimento: [{value: '', disabled: true}, Validators.required]
```

Se formos executar a aplicação no navegador, veremos que nada ocorre ao clicarmos em qualquer parte do `<ion-item>`, pois o componente é nativo e precisa estar executando em um emulador ou dispositivo.

Mas e se quisermos ter, para os dispositivos, o controle nativo e, para o navegador, o controle Ionic? Isso é possível e tranquilo. Precisaremos trabalhar com a identificação da plataforma onde a aplicação está sendo executada e veremos isso agora.

Nosso primeiro passo é injetar em nosso componente um serviço que nos possibilitará a identificação da plataforma em que o aplicativo está sendo executado. Insira no construtor, entre os argumentos, a instrução a seguir. Será preciso adicionar `Platform` ao import do `@ionic/angular`.

```
private platform: Platform
```

Agora precisamos, antes de invocar o método do `DatePicker`, verificar a plataforma e, se for a desejada, executamos o código que trabalhamos há pouco para a renderização do controle. Veja o novo código para nosso método `selecionarData()` na sequência.

```
selecionarData() {
    this.platform.ready().then(() => {
      if (this.platform.is('capacitor')) {
        this.datePicker.show({
          date: new Date(),
          mode: 'date',
          locale: 'pt-br',
          doneButtonLabel: 'Confirmar',
          cancelButtonLabel: 'Cancelar'
        })
```

```
        .then(
          data =>
this.clienteForm.controls.nascimento.setValue(data.toLocaleDateString())
        );
      } else {
        // instruções para execução no navegador
      }
    });
}
```

Veja que agora o início do método é a execução de uma `Promise` que, ao ser concluída, verificamos a plataforma em execução e então inserimos nosso já conhecido código. Deixei o `else` para sabermos dessa possibilidade. Se quiser verificar as plataformas possíveis de serem identificadas, acesse https://ionicframework.com/docs/angular/platform. Tente executar sua aplicação no navegador agora. Clique sobre o `<ion-item>` .

Ainda temos nosso `<ion-datetime>` desabilitado mesmo executando no navegador. Precisamos agora condicionar este estado do componente à plataforma em execução. Em nosso componente, vamos implementar a seguinte propriedade de leitura.

```
get isBrowserPlatform(): boolean {
    if (this.platform.is('capacitor')) {
      return false;
    }
    return true;
}
```

Em situações diferentes, trabalhamos a diretiva `*NgIf` e, para efeito didático em um novo conhecimento, vamos usá-la agora para resolver essa nossa situação, mas vamos utilizá-la com um `else` . Nosso objetivo é que, caso estejamos com a aplicação em uso em um dispositivo, o componente nativo seja renderizado, caso estejamos no navegador, o `<div>` clicável seja o componente renderizado. Veja então esta implementação na sequência.

```
<div *ngIf="isBrowserPlatform; else showNative">
      <ion-grid>
        <ion-row>
          <ion-col size="9">
            <ion-item lines="none">
              <ion-label position="floating">Nascimento</ion-label>
              <ion-input [value]="dataSelecionada" [readonly]="true">
</ion-input>
            </ion-item>
          </ion-col>
          <ion-col class="ion-no-padding">
            <ion-item lines="none">
              <ion-button fill="clear" id="open-date-input" class="ion-no-
margin;ion-no-padding">
                <ion-icon icon="calendar"></ion-icon>
              </ion-button>
              <ion-popover trigger="open-date-input" show-backdrop="false"
[dismissOnSelect]="true">
                <ng-template>
                  <ion-datetime #popoverDatetime presentation="date"
formControlName="nascimento"
                    displayFormat="DD/MM/YYYY" min="1967" max="2022-12-31"
                    (ionChange)="dataSelecionada = confirmarData()">
                  </ion-datetime>
                </ng-template>
              </ion-popover>
            </ion-item>
          </ion-col>
        </ion-row>
      </ion-grid>
</div>

<ng-template #showNative>
      <div (click)="selecionarData()">
        <ion-grid class="ion-no-padding">
          <ion-row>
            <ion-col size="9">
              <ion-item lines="none">
                <ion-label position="stacked">Nascimento</ion-label>
```

```
                <ion-input formControlName="nascimento"></ion-input>
              </ion-item>
            </ion-col>
            <ion-col class="ion-no-padding">
              <ion-item lines="none">
                <ion-button fill="clear" class="ion-no-margin;ion-no-
padding">
                  <ion-icon icon="calendar"></ion-icon>
                </ion-button>
              </ion-item>
            </ion-col>
          </ion-row>
        </ion-grid>
      </div>
</ng-template>
```

No início do código anterior, temos um `<div>` comum do HTML, mas que será renderizado apenas caso o método `get isBrowsePlatform` retorne verdadeiro, caso contrário, um template chamado `showNative` será renderizado. Esse template está logo após o primeiro `<div>`.

Verifique o uso de `<ng-template>` e a maneira de declararmos um identificador para ele (`#showNative`). Temos nele um novo `<div>` com os componentes necessários para esta situação. Note que o método `selecionarData()` está ligado ao `click` do `<div>`.

Precisamos alterar o código que configura o controle `nascimento` no `Reactive Forms`, pois a desabilitação de nosso controle agora depende de verificação da plataforma. Veja a adaptação na sequência e teste sua aplicação.

```
nascimento: [{value: '', disabled: !this.isBrowserPlatform },
Validators.required]
```

Vamos testar a aplicação no emulador ou dispositivo físico e ver como se comporta o componente para a data de nascimento? Para ajudar, trago na sequência a instrução a ser executada. Lembre-se de iniciar o emulador ou conectar seu dispositivo físico. Veja como

será exibido o componente para a data de nascimento. Em caso de erro de conexão, lembre-se do `--external`.

```
// Preparação/construção do projeto
npm run build

// Execução no iOS
ionic capacitor run ios -l

// Execução no Android
ionic capacitor run android -l
```

3.12 Confirmando a inserção com o usuário

É sempre interessante uma boa interação com o usuário de nossos aplicativos, e informarmos a ele o sucesso ou não de uma operação é extremamente relevante. Desta maneira, fechando este capítulo, trabalharemos alguns recursos do Ionic para produzir informações de resultados de operações para o usuário.

O componente `Alert` é responsável pela renderização de um tipo de janela no dispositivo, com título, subtítulo, mensagem e botões. Existem vários recursos para o componente que podem ser utilizados e trabalharemos aqui algumas dessas técnicas.

Precisamos injetar o `Alert` no construtor de nosso componente para então o utilizarmos. Será preciso importar o `AlertController`. Para facilitar, no código a seguir também está o `import` necessário.

```
import { AlertController } from '@ionic/angular';

private alertCtrl: AlertController
```

Com o `Alert` disponibilizado para nosso componente, criaremos um método responsável pela geração da mensagem de alerta ao usuário para então o consumirmos.

```
async presentAlert(header: string, subHeader: string, message: string,
buttons: string[]) {
    const alert = await this.alertCtrl.create({
      header,
      subHeader,
      message,
      buttons
    });
    await alert.present();
}
```

Observe a assinatura do nosso método. Veja o uso da instrução `async`, que torna nosso método assíncrono. Isso possibilita a continuidade de execução da aplicação enquanto o método não é concluído, evitando o sentimento de "travamento" da aplicação.

Os argumentos recebidos pelo método são os necessários para a criação de uma mensagem básica e seus nomes são semânticos, o que facilita nossa compreensão. Observe que a criação e a apresentação da mensagem são precedidas da instrução `await`, que trabalha em conjunto com o `async`. Sempre que temos chamadas `await` em um método, o método que engloba a instrução precisa ser declarado como `async`. Note que, como os argumentos recebidos têm o mesmo nome das propriedades do objeto enviado para `create()`, basta referenciá-los sem precisar de uma atribuição.

Resumindo o método: recebemos os argumentos, os enviamos ao método `create()` do nosso objeto `alertCtrl` e, com o objeto devidamente criado, invocamos a apresentação pela chamada a `present()` de nosso objeto `alert`. Note que declaramos `alert` como `const`, pois não mudaremos o valor dela.

Resta-nos invocar nosso novo método e faremos isso na submissão de nosso formulário, no `submit()`. Veja a implementação e observe que precisamos alterá-lo para `async` e que a chamada ao método `presentAlert()` tem o `await`. Observe que retornei ao código o `errorsMessage` e `hasErrors` para esse teste.

```
async submit() {
    this.errorsMessage = [];
    if (this.clienteForm.get('nome').hasError('required') ||
this.clienteForm.get('nome').hasError('minLength')) {
      this.errorsMessage.push('Nome é obrigatório');
    }
    if (this.clienteForm.get('email').hasError('required') ||
this.clienteForm.get('email').hasError('email')) {
      this.errorsMessage.push('Email é obrigatório');
    }
    this.hasErrors = this.errorsMessage.length > 0;

    if (!this.hasErrors) {
      await this.presentAlert('Sucesso', 'Gravação bem sucedida', 'Os
dados do cliente foram gravados', ['Ok']);
    }
}
```

Vamos testar nossa aplicação. Preencha os dados do formulário e pressione o botão. Sua janela deve estar semelhante à apresentada na sequência.

12:34 PM
100%
Registro de CLIENTES
Nome
Ambrósio
Email
ambrosio@cc.com.br
Telefone
4512
Rend
23
Nasci
01/0
Sucesso
Gravação bem sucedida
Os dados do cliente foram gravados
OK
CONFIRMAR

Figura 3.7: Exibindo mensagens por meio do Alert.

O Ionic traz outro recurso para interação com o usuário no que diz respeito à exibição de mensagens. É muito comum recebermos em nossos dispositivos, principalmente no topo da tela, mensagens conhecidas como notificações. Estas notificações são conhecidas também como `Toast`.

Assim como fizemos para o `Alert`, precisamos injetar em nosso componente o `ToastController`. Sendo assim, insira ao final do nosso construtor a instrução a seguir. Ajuste a importação.

```
private toastCtrl: ToastController
```

Na sequência, vamos implementar um método que será responsável pela apresentação de nossa mensagem ao usuário. Veja o código para ele na sequência.

```
async presentToast(message: string, duration: number, position: 'top' |
'bottom') {
    const toast = await this.toastCtrl.create({
      message,
      duration,
      position
    });
    toast.present();
}
```

A assinatura do método traz alguns argumentos básicos para a exibição do controle: mensagem, duração em milissegundos, e a posição para a exibição da mensagem. Observe que o método também é assíncrono e a criação do componente é precedida pelo `await`. Ainda na assinatura, veja que limitamos os valores possíveis de serem recebidos para a posição do controle a `top` e `bottom`.

Para testarmos nossa nova implementação no `submit()`, vamos trocar a invocação ao `Alert` para o `Toast`. Veja esta implementação na sequência e finalizamos então nossa seção e capítulo.

```
async submit() {
    // Regras de validação
    await this.presentToast('Gravação bem sucedida', 3000, 'top');
}
```

Conclusão

Terminamos o capítulo com nossa primeira implementação de uma funcionalidade! Já vimos e trabalhamos bastante coisa. Conhecemos a estrutura básica de uma aplicação Ionic e as definições sobre como os arquivos (artefatos) são gerados e ligados ao padrão de componentes trabalhado pelo framework. Pudemos interagir com o Ionic CLI para a criação da nossa página, que é vista como um componente em aplicações Ionic. Na página que criamos, inserimos componentes de entrada e saída de dados e vimos possibilidades de uso de teclados diferentes para cada tipo de dado a ser informado. Configuramos alguns controles diretamente com diretivas Ionic e fizemos uso de classes de estilo (CSS).

Descobrimos como agrupar controles por meio do componente `<ion-item>` e implementamos a classe do componente para acessar os valores informados nos controles de entrada. O Ionic oferece recursos para ligação de dados e eventos (data binding e event binding) e pudemos colocar isso em prática também neste capítulo. Finalizando o processo de entrada de dados, trabalhamos o `Angular Reactive Forms`, que traz, dentre seus recursos, possibilidades para validações sobre os dados informados e emissão de alertas ao usuário para o caso de erro na entrada.

Dentre os tipos de dados que trabalhamos, tínhamos a data de nascimento, o que nos levou a conhecer um componente nativo para esse tipo de dado, promovendo assim a experiência do usuário em seu dispositivo e recursos visuais oferecidos por sua plataforma. Finalizando o capítulo, trabalhamos dois componentes que exibem mensagens ao usuário a fim de informá-lo do sucesso no processo de gravação.

Foi um capítulo introdutório com muitos recursos. Podemos agora dar uma relaxada, tomar uma água e então voltarmos para o próximo capítulo, no qual trabalharemos a exibição de valores em forma de listagem.

Capítulo 4
Listagem de dados e navegação entre páginas

No capítulo anterior, vimos vários recursos para a entrada de dados por parte do usuário. Não chegamos a persistir os dados informados, pois veremos isso no próximo capítulo, mas agora poderemos verificar as possibilidades de exibição de dados armazenados localmente em uma matriz e depois uniremos todos esses recursos atrelados à persistência e à recuperação de dados.

Nesta situação, vou optar pela criação de um novo projeto até para podermos praticar essas funcionalidades. Entretanto, caso queira, você pode dar continuidade ao projeto anterior.

A funcionalidade que implementaremos aqui está ligada ao registro dos tipos de serviços que a oficina pode prestar, consulta e alteração de tipos de serviços já registrados. Introduziremos também os componentes de serviço (services) oferecidos pelo Ionic/Angular.

4.1 Criação e execução inicial do projeto

Em sua pasta de projetos Ionic, como sugestão, insira no terminal a seguinte instrução. Veja que nosso projeto agora se chama `capitulo04`.

```
ionic start capitulo04 blank --type=angular --no-git
```

No Visual Studio Code, com a pasta do projeto `capitulo04` aberta, vamos criar uma nova funcionalidade para o aplicativo, tal qual fizemos no capítulo anterior. Veja a instrução a seguir, agora focada em uma listagem para tipos de serviços. Note que, diferentemente

do capítulo anterior, onde criamos as páginas diretamente em `app`, optamos agora por uma pasta específica para `pages`.

```
ionic generate page pages/tipo-servicos/tipo-servicos-listagem
```

Já sabemos que, ao executarmos nossa aplicação, a página que será exibida é a `Home`. Mas sabemos como corrigir isso, certo? Vamos abrir o arquivo `app-routing.module.ts` e atualizar a rota com `path: ''`, para que o `redirectTo`, seja para `tipo-servicos-listagem`.

Se você quiser, pode testar sua aplicação agora, mas saiba que nada será exibido ainda, a não ser o título da página, que foi criado pelo `generate`. Lembre-se de que na primeira execução, se você for executar o `ionic serve --lab`, será solicitada a instalação do `@ionic/lab`. Caso você queira, podemos realizar a instalação com o código a seguir antes de executar a aplicação.

```
npm install @ionic/lab
```

4.2 Populando a página com itens

Sempre que desenvolvemos uma aplicação que manipula dados informados pelo usuário, normalmente fazemos uso de algum mecanismo de persistência para que esses dados estejam disponíveis entre as sessões de uso da aplicação. No capítulo anterior, nós apenas trabalhamos com a interface com o usuário, que solicitava esses dados, não os armazenamos. Neste capítulo, nós trabalharemos com a exibição desses dados, mas ainda sem a persistência física deles. Faremos uso de arrays, pois nosso foco aqui ainda é a interface com o usuário.

Em nosso componente, precisamos criar e disponibilizar uma matriz com os dados que queremos exibir ao usuário. No código a seguir, veja a matriz que devemos implementar logo no início da classe do nosso componente. Observe o modificador de escopo `public`, que

possibilitará o consumo dessa variável pelo nosso template. Fique à vontade para inserir mais itens para ver a página totalmente ocupada.

```
public tiposServicos = [
    { id: 1, nome: 'Alinhamento', valor: 12.34 },
    { id: 2, nome: 'Balanceamento', valor: 56.78 },
    { id: 3, nome: 'Revisão freios', valor: 90.12 },
    { id: 4, nome: 'Suspensão', valor: 34.56 }
];
```

Com a fonte de dados criada, precisamos agora renderizar esses dados em nosso template e faremos isso com o `<ion-list>`, que utilizamos no capítulo anterior para exibir os erros no início da página. Veja o código a seguir. Observe a junção de componentes Ionic com HTML e CSS.

```
<ion-header>
  <ion-toolbar>
    <ion-title>Tipos de serviços</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content padding>
  <ion-list>
    <ion-item *ngFor="let tipoServico of tiposServicos">
        <ion-label>
            <h2>{{tipoServico.nome}}</h2>
            <p style="text-align:right">R$ {{tipoServico.valor}}</p>
        </ion-label>
    </ion-item>
  </ion-list>
</ion-content>
```

A figura a seguir apresenta os pseudoemuladores com a aplicação em execução. Nesta figura, trago mais itens do que temos registrado em `tiposServicos`, ok?

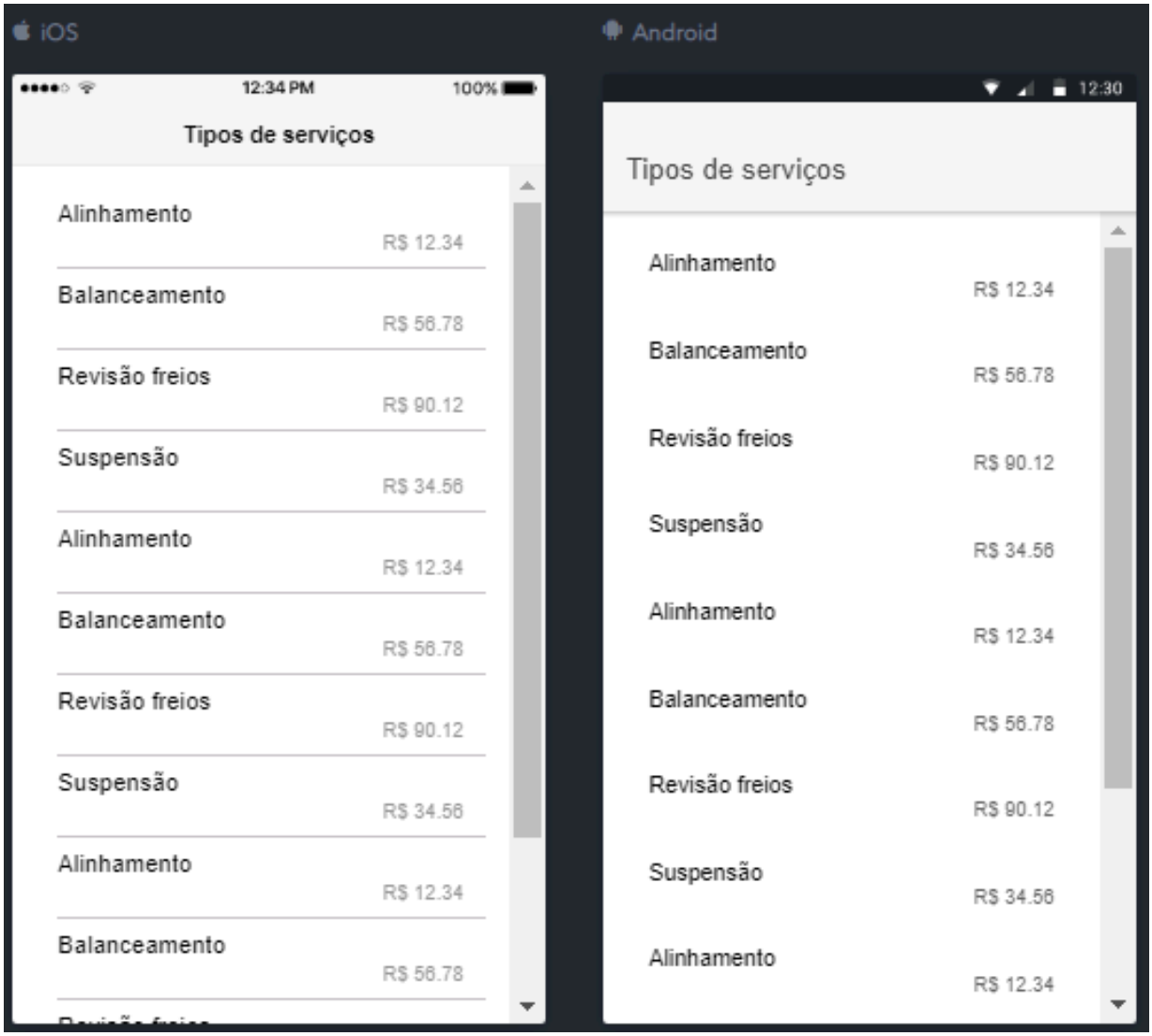


Figura 4.1: Listagem de tipos de serviços.

4.3 Navegação para uma página de detalhes de um tipo de serviço

É muito comum que, a partir de uma visão onde os dados são listados, seja possível a navegação para uma página com maiores detalhes sobre um determinado item selecionado em tal listagem.

Da mesma maneira que criamos a página para listagem, vamos criar nossa página de inserção e alteração. Daremos aos artefatos um nome que semanticamente abrangerá a inserção e alteração de um tipo de serviço. Sendo assim, execute no terminal, na pasta do projeto, a instrução a seguir e já vamos partir para as implementações dos arquivos gerados e ajustes imediatos.

```
ionic generate page pages/tipo-servicos/tipo-servicos-add-edit
```

Arquivo tipo-servicos-add-edit.page.html

No código a seguir, observe que temos um novo controle, o `<ion-item-divider>` . Esse controle renderiza um tipo de contêiner na página destacando os controles que estão dentro dele. Fazemos uso do sistema de grids do Ionic no `<ion-item-divider>` e, em nossa primeira linha, temos uma coluna com uma imagem por meio do componente `<ion-img>` . Observe o caminho para a figura. A pasta `assets` está no mesmo nível de `app` e você precisará criar a `imgs` e copiar a figura (que pode ser a que quiser) para esta pasta.

```
<ion-header>
    <ion-toolbar>
        <ion-title>Tipos de serviços</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content padding>
    <ion-item-divider color="light">
        <ion-grid no-padding>
            <ion-row no-padding>
                <ion-col col-3 align-self-center>
                    <ion-img style="width: 40px; height: 40px;"
src="assets/imgs/tab_servicos.png"></ion-img>
                </ion-col>
                <ion-col col-9 align-self-center>
                    <ion-label style="font-size: 23px">Dados do
serviço</ion-label>
                </ion-col>
```

```
            </ion-row>
        </ion-grid>
    </ion-item-divider>
</ion-content>
```

Como utilizamos o `ionic generate`, a inclusão da rota no `app-routing.module.ts` já foi realizada, mas precisamos adaptá-la, pois esta nossa nova rota receberá o `id` do item que será exibido. Veja a alteração em `path`, na listagem a seguir.

```
{
    path: 'tipo-servicos-add-edit/:id',
    loadChildren: () => import('./pages/tipo-servicos/tipo-servicos-add-edit/tipo-servicos-add-edit.module').then(m => m.TipoServicosAddEditPageModule)
},
```

Mas como chegar à rota especificada anteriormente? É bem simples, vamos apenas adicionar ao `<ion-item>` da listagem (no template) a instrução a seguir. Observe o binding para `tipoServico`, que é a variável inicializada a cada iteração do `*ngFor`. Isso deve ser atualizado no arquivo `tipo-servicos-listagem.page.html`.

```
<ion-item *ngFor="let tipoServico of tiposServicos" routerLink='/tipo-servicos-add-edit/{{tipoServico.id}}'>
```

Já podemos testar nossa aplicação. Na listagem, selecione um tipo de serviço. Você será redirecionado para a página de detalhes e verá que nossos dados ainda não aparecem. Logo veremos isso. Temos agora um novo problema: como retornar para a página anterior? Em nosso template para a página de detalhes, logo no início, dentro do `<ion-toolbar>` e antes do `<ion-title>`, insira o código a seguir, lembrando que é no arquivo `tipo-servicos-add-edit.page.html`.

```
<ion-buttons slot="start">
    <ion-back-button defaultHref=""></ion-back-button>
</ion-buttons>
```

No código anterior, temos o surgimento de `slot`, que é uma característica relacionada a `Web Components`. O `slot` refere-se ao posicionamento de um conteúdo dentro de um contêiner que em nosso caso é a barra de tarefas.

Voltando ao código, observe o componente `<ion-back-button>`. Esse componente define como URL de destino uma string vazia que, ao retornar ao `app-routing.module.ts`, você verá que é uma rota que redireciona a requisição para nosso `home`, que se refere à nossa listagem. Vamos testar novamente nossa aplicação? Caso a navegação não funcione, lembre-se do F5 ou CTRL+F5 para atualizar a página.

4.4 Recuperação do tipo de serviço selecionado

Já temos nossa página de listagem com os itens apontando para uma página de detalhes e enviando o `id` do tipo de serviço que deve ser exibido, mas, como exibir os dados específicos para o `id` recebido por essa página?

Nossa listagem é populada por um objeto que representa uma matriz e que está definida no componente de listagem, então nossa página de detalhes não conseguirá pesquisar essa matriz a não ser que a enviemos como parâmetro para a página, mas não é isso o que faremos. Faremos uso de serviços (*services*) e injeção de dependência (*dependency injection*).

O Angular e, consequentemente, o Ionic oferecem um tipo de componente chamado `Services`, que são classes que podem ter objetos injetados em nossos componentes. Para nosso problema, temos a situação de que precisamos de um conjunto de dados, que no momento é uma matriz, mas poderia ser uma base de dados ou dados fornecidos por serviços REST.

Vamos então trazer para a nossa aplicação a criação de um serviço que oferecerá acesso aos dados que deverão ser consumidos. O primeiro passo é criarmos, na pasta `app`, uma nova pasta chamada `services` e, nela, um arquivo chamado `tipo-servicos.service.ts`. Com o arquivo criado, vamos implementar nele o seguinte código.

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class TipoServicosService {

  private tiposServicos = [
    { id: 1, nome: 'Alinhamento', valor: 12.34 },
    { id: 2, nome: 'Balanceamento', valor: 56.78 },
    { id: 3, nome: 'Revisão freios', valor: 90.12 },
    { id: 4, nome: 'Suspensão', valor: 34.56 }
  ];

  constructor() { }

  getById(id: number) {
    const tipoServicoSelecionado = this.tiposServicos.filter(
      tipoServico => tipoServico.id === id
    );
    return tipoServicoSelecionado[0];
  }
}
```

Note que o serviço nada mais é que uma classe comum que em nosso caso será visto também como um `provider`, por isso a existência de um decorator chamado `@Injectable`. Como nossa intenção é prover esse serviço para toda a aplicação, enviamos como argumento para o decorator o elemento `providedIn` com o valor `root`. Isso gerará uma única instância do serviço a ser compartilhada por toda a aplicação. Se fôssemos prover o serviço para um módulo específico, iríamos declará-lo em sua especificação.

Observe também a declaração de nossa já conhecida matriz, pois queremos que os dados agora sejam fornecidos por uma única fonte. Após o construtor, temos a definição de um método que receberá um `id`. Este valor será utilizado para filtrar um elemento que coincida em valor e tipo (`===`) com a propriedade `id` de um elemento da matriz. Ao final, retornamos o elemento selecionado. Caso nada seja encontrado, `undefined` é atribuído ao elemento.

Apenas para fins de conhecimento, outra maneira de criar o componente `service` seria por meio do Ionic CLI com a instrução `ionic generate service services/tipo-servicos.service`.

Precisamos agora ajustar nosso componente `tipo-servicos-add-edit.page.ts` para receber nosso serviço e então poderemos consumir o método que implementamos nele. Veja o novo código para o corpo da classena sequência. Algumas importações serão necessárias.

```
export class TipoServicosAddEditPage implements OnInit {
  public tipoServico;

  constructor(
    private route: ActivatedRoute,
    private tipoServicoService: TipoServicosService
  ) { }

  ngOnInit() {
    const id: number = Number(this.route.snapshot.paramMap.get('id'));
    this.tipoServico = this.tipoServicoService.getById(id);
  }
}
```

Antes do construtor, há a definição de uma propriedade chamada `tipoServico`. Ela conterá o tipo de serviço selecionado na página de listagem e terá seus dados consumidos pelo template de exibição dos detalhes que estamos implementando. Na sequência, temos nosso construtor, recebendo um objeto para tratamento de rotas (`ActivatedRoute`) e nosso serviço anteriormente implementado (`TipoServicosService`).

Finalizando o código, temos a implementação para o `ngOnInit()`, que recupera o parâmetro enviado pela página de listagem e o utiliza para recuperar o objeto completo que é então atribuído à nossa propriedade para que os dados possam ser exibidos corretamente pelo template. Não nos preocupamos em verificar se um dado foi recuperado corretamente, pois, em nosso caso, o dado foi exibido, então temos a garantia de sua existência, mas é algo em que devemos pensar, talvez verificando e validando o objeto retornado ou trabalhando com exceções no método de serviço - veremos isso adiante.

Para testarmos, insira no arquivo HTML o código a seguir, abaixo de `</ion-item-divider>` e, em seguida, selecione um serviço na listagem e vamos agora ver nossa página de detalhes com este tipo de serviço devidamente renderizada. Esses valores serão recuperados por meio do binding das `{{}}`. Eu precisei recarregar a página para que funcionasse em meu teste.

```
<ion-item>{{tipoServico.nome}}</ion-item>
<ion-item>
    <span item-end>R$ {{tipoServico.valor}}</span>
</ion-item>
```

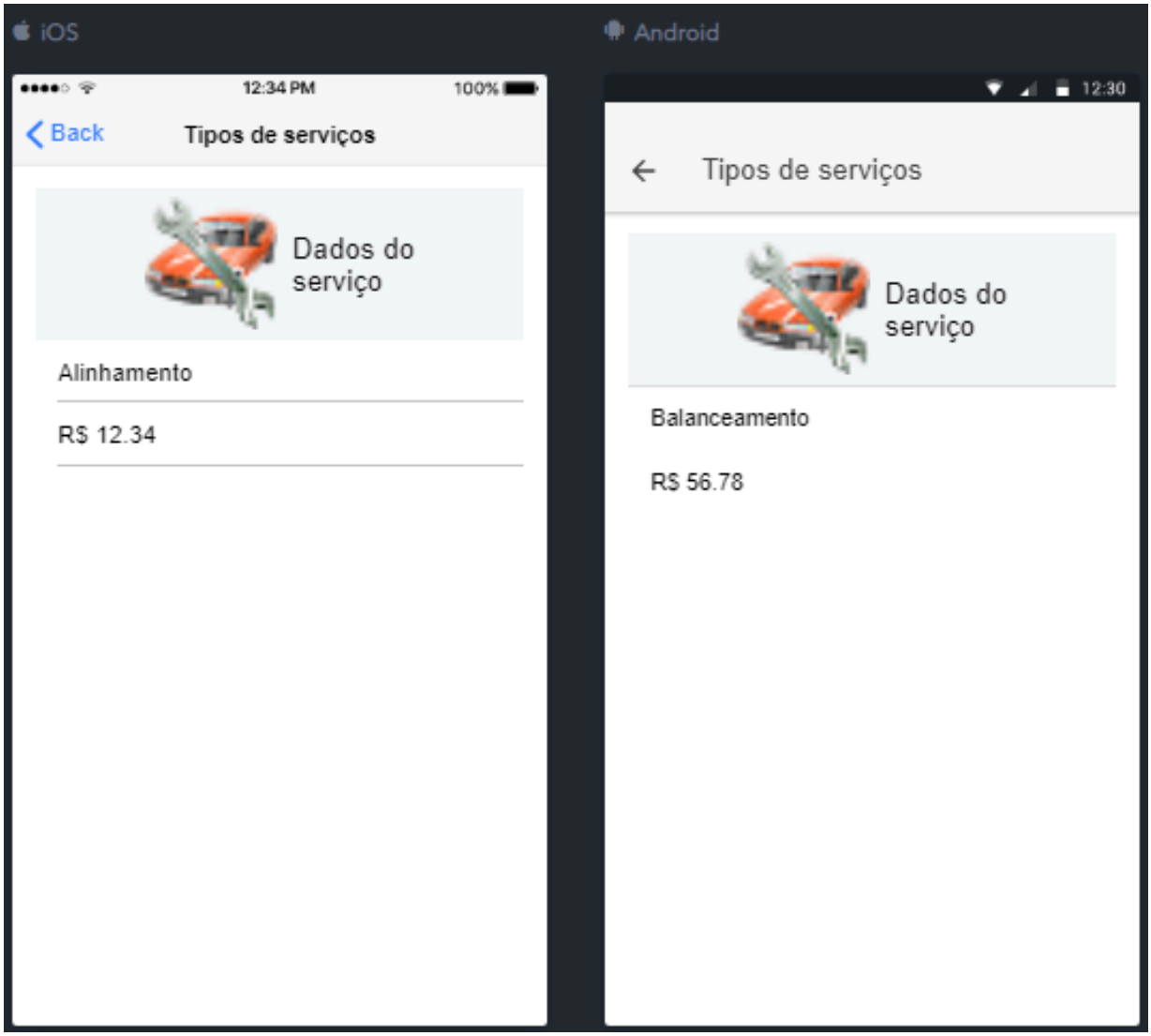


Figura 4.2: Detalhes de um item selecionado na listagem.

4.5 Alteração dos dados exibidos na página de detalhes

Já temos nossa página de detalhes exibindo corretamente os dados de um tipo de serviço selecionado pelo usuário em uma página de listagem. Precisamos agora oferecer ao nosso usuário a possibilidade de alterar esses dados. Faremos uso de um botão para permitir essa

funcionalidade e teremos algumas mudanças também em nosso código de componente e template. Nosso primeiro passo será a inserção do botão que habilitará a alteração. Veja o código a seguir, que deve ser inserido no arquivo `tipo-servicos-add-edit.page.html` antes da tag de fechamento `<ion-content>`.

```
<ion-button *ngIf="!modoDeEdicao" shape="round" color="primary"
expand="block" padding (click)="iniciarEdicao()">
    Alterar dados
</ion-button>
```

O componente relativo a botões não é novo para nós, já o vimos nos capítulos anteriores. Tirando os atributos de configuração, temos a diretiva condicional do Angular, `*ngIf`, que determinará quando o controle será renderizado e podemos verificar que isso ocorrerá na negação de uma variável chamada `modoDeEdicao`, que já declararemos. Ao final da tag `<ion button>` de abertura, temos um `Event Bind` para `click` para um método que também não implementamos ainda.

Em nosso arquivo do componente, inclua a instrução a seguir antes da assinatura do construtor e, após o construtor, o método que também está na listagem.

```
// Declaração de propriedade para sinalizar modo de edição e consulta
public modoDeEdicao = false;

// Método que registrará que o tipo de serviço estará em processo de
edição
iniciarEdicao() {
    this.modoDeEdicao = true;
}
```

Execute sua aplicação e veja a exibição do botão. Clique/pressione o botão e veja que ele desaparece. É este o caminho que queremos seguir.

Utilizamos o binding por meio de `{{}}` com nosso componente para então termos o dado exibido ao usuário. Entretanto, agora não

queremos essa página apenas para exibir dados, queremos que o usuário tenha a possibilidade de alterar os dados nela. Isso nos leva à necessidade de um formulário e componentes que renderizem controles de interação. Precisamos reformular nosso template tal qual segue na listagem para o arquivo `tipo-servicos-add-edit.page.html`, abaixo de `</ion-item-divider>`, substituindo o código existente.

```
<form [formGroup]="tiposServicosForm" (submit)="submit()">
    <ion-item lines="none" [disabled]="!modoDeEdicao">
        <ion-label position="floating" class="label-input-
title">Nome</ion-label>
        <ion-input formControlName="nome" type="text" style="margin-left:
3%;"></ion-input>
    </ion-item>

    <ion-item lines="none" [disabled]="!modoDeEdicao">
        <ion-label position="floating" class="label-input-
title">Valor</ion-label>
        <ion-input text-right formControlName="valor" type="number"></ion-
input>
    </ion-item>

    <ion-button *ngIf="!modoDeEdicao" shape="round" color="primary"
expand="block" padding (click)="iniciarEdicao()">
        Alterar dados
    </ion-button>

    <div *ngIf="modoDeEdicao" no-padding>
        <ion-grid>
            <ion-row>
                <ion-col col-6 >
                    <ion-button shape="round" color="success"
                                size="small" padding expand="block"
                                type="submit">
                        Gravar
                    </ion-button>
                </ion-col>
```

```
                <ion-col col-6>
                    <ion-button shape="round" color="warning"
                                size="small" padding expand="block"
                                (click)="cancelarEdicao()">
                        Cancelar
                    </ion-button>
                </ion-col>
            </ion-row>
        </ion-grid>
    </div>
</form>
```

Logo no início do código anterior, temos a definição de nosso formulário, que é um Reactive Form. Temos um nome ligado à propriedade `formGroup`, um evento ligado ao `submit` e depois dois `<ion-item>` apresentando os dados do tipo de serviço selecionado, já com controles que possibilitem a alteração dos dados. Mas atenção: veja nos `<ion-item>` a definição e ligação do `disabled` com nossa variável de controle. Com isso, os controles só aceitarão interação quando clicarmos no botão para alterar os dados.

Após o botão que habilita a alteração de dados, temos uma `<div>`, que será exibida apenas quando estivermos alterando um tipo de serviço, pois ela contém dois botões: um que realizará a gravação e será o botão de submissão do formulário e um que registrará o cancelamento dos dados alterados. Veja o uso de grids para os botões.

Precisamos agora declarar em nosso arquivo do módulo (o `tipo-servicos-add-edit.module.ts`) a importação de `ReactiveFormsModule`, em `imports`. Depois, precisamos definir nosso `FormGroup`, que estamos usando em nossa tag `<form>`. E essa é a instrução da listagem a seguir, que deve ser implementada antes do construtor do arquivo `tipo-servicos-add-edit.page.ts`. Será preciso importar `FormGroup`, mas o Visual Studio Code ajuda você nisso.

```
public tiposServicosForm: FormGroup;
```

Dando sequência, na preparação do nosso componente para o Reactive Form, precisamos injetar a instrução a seguir em nosso construtor. Precisaremos também importar `FormBuilder`.

```
private formBuilder: FormBuilder
```

Vamos à adaptação de nosso método `ngOnInit()`, tal qual segue. Já temos implementadas as duas primeiras instruções, mas preferi mantê-las na listagem. Observe que, após termos recuperado o objeto selecionado na listagem, configuramos nosso `FormGroup` criando suas propriedades e atribuindo a elas o valor inicial, com base no objeto recuperado. Definimos também uma validação básica. Atente para a importação para `Validators`.

```
ngOnInit(): void {
    const id: number = Number(this.route.snapshot.paramMap.get('id'));
    this.tipoServico = this.tipoServicoService.getById(id);

    this.tiposServicosForm = this.formBuilder.group({
        id,
        nome: [this.tipoServico.nome, Validators.required],
        valor: [this.tipoServico.valor, Validators.required]
    });
}
```

Execute a aplicação, selecione um item da listagem e veja os controles renderizados e desabilitados. Clique/pressione o botão que habilita a alteração, observe a ocultação deste botão e exibição dos outros dois, assim como a habilitação dos controles de interação. Compare sua aplicação com a figura a seguir.

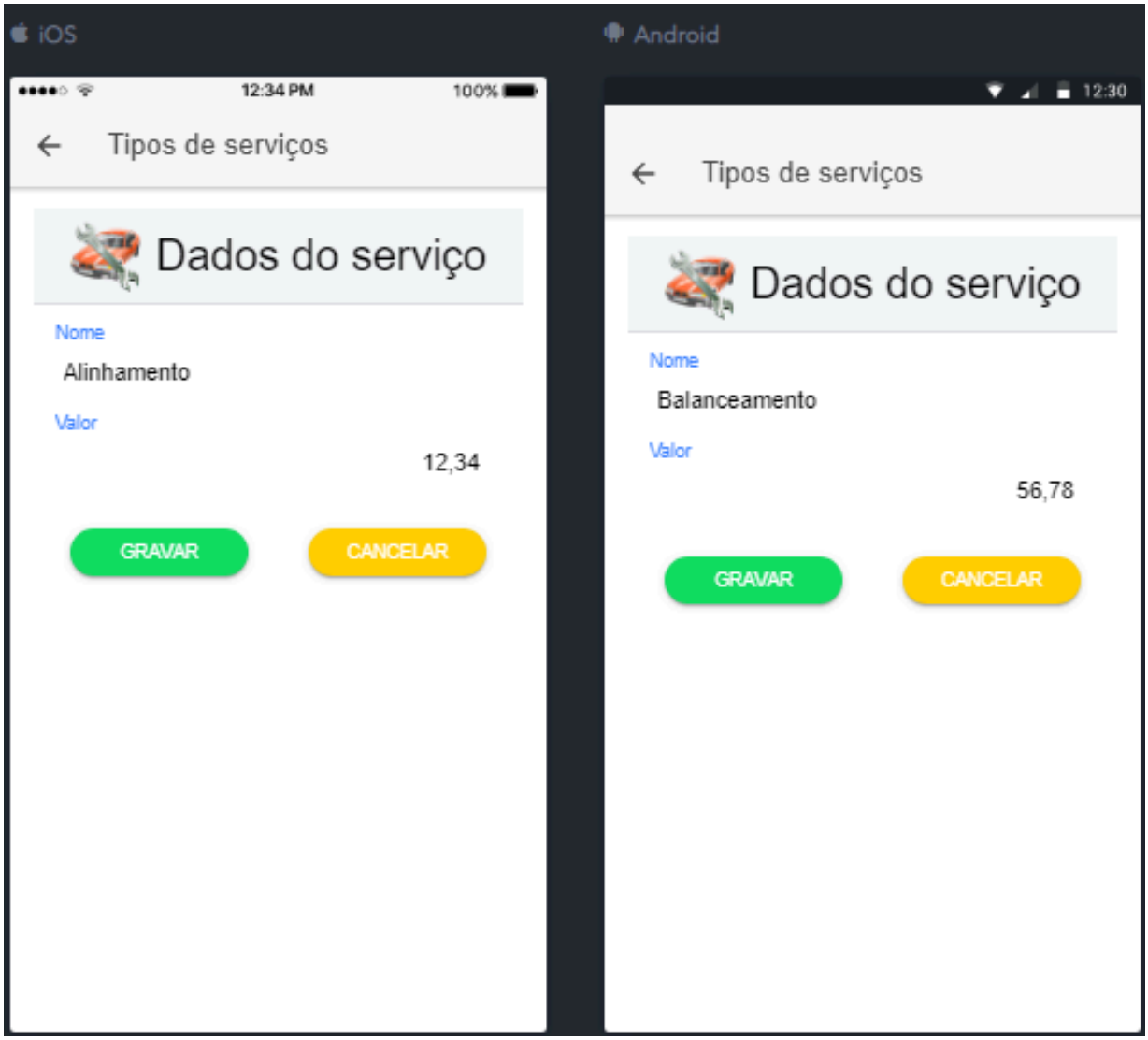


Figura 4.3: Alteração de um tipo de serviço.

Com a interface com o usuário pronta, precisamos realizar a atualização dos dados alterados. Em nosso código anterior, lembra que fizemos um binding com `submit` na tag `<form>` para o template? Pois é, precisamos agora implementar nossos serviços que deverão ser atendidos por essa operação. Começaremos com a implementação dos métodos em nossa classe de serviço, onde implementamos nossa matriz e o método `getById()`. Veja os novos métodos no arquivo `tipo-servicos.service.ts`.

```
private getIndexOfElement(id: number): number {
    return this.tiposServicos.indexOf(this.getById(id));
}

update(tipoServico: any) {
    this.tiposServicos[this.getIndexOfElement(tipoServico.id)] =
tipoServico;
}
```

O primeiro método da listagem anterior nos retornará o índice do elemento referente ao `id` que o método recebe. Como estamos lidando com matrizes, este é o caminho para atualizarmos. Já o segundo método, `update()`, atualizará o elemento com o índice recuperado pelo método `getIndexOfElement()`. Agora sim vamos ao método `submit()` em nosso arquivo TypeScript da página de edição. Veja o código na sequência.

```
submit() {
    this.tipoServicoService.update(this.tiposServicosForm.value);
    this.modoDeEdicao = false;
}
```

Observe que enviamos o valor atual de nosso `FormGroup` para o método `update()` de nosso serviço, pois os dados informados pelo usuário estão nos componentes deste objeto. Após a execução da atualização, alteramos o estado de nossa página para apenas visão por meio de nossa variável controladora `modoDeEdicao`.

Agora já podemos realizar a alteração e gravar os novos dados. Na aplicação, após gravar os novos dados, volte para a listagem. Verifique que ela não mostra a atualização registrada.

Abra o TypeScript de nosso componente de listagem. Veja que, no início deste capítulo, tínhamos definido uma matriz e que o nosso template está baseado nessa coleção. Você se lembra de que, para disponibilizarmos os dados do item selecionado na listagem, criamos um serviço e disponibilizamos neste serviço uma matriz que será de uso comum em toda a aplicação? Pois bem, estamos assim exibindo

uma coleção na listagem e trabalhando em cima de outra coleção na alteração. Retire a matriz e deixe apenas a declaração a seguir no componente de listagem.

```
public tiposServicos;
```

Inicializaremos a propriedade anterior com base em dados obtidos em nosso serviço, o que nos leva a injetar esse serviço em nosso componente. Desta maneira, insira a seguinte declaração no construtor do nosso componente. Precisaremos importá-lo.

```
constructor(
        private tipoServicoService: TipoServicosService
    ) { }
```

Agora precisamos trabalhar em nosso serviço, pois ele não nos oferece nada para recuperar todos os tipos de serviços existentes. Sendo assim, no arquivo `tipo-servicos.service.ts` , adicione o seguinte método.

```
getAll() {
    return this.tiposServicos;
}
```

Vamos retornar para nosso componente e consumir este serviço e faremos isso como já de costume para processos relacionados à inicialização do componente, no método `ngOnInit()` . Veja a implementação a seguir, que já se traduz por si só. O arquivo a implementar é o `tipo-servicos-listagem.page.ts` .

```
ngOnInit() {
    this.tiposServicos = this.tipoServicoService.getAll();
}
```

Execute sua aplicação, selecione um item na listagem, altere-o, grave a alteração e retorne para a listagem. Veja que ela agora se apresenta atualizada, pois a coleção de objetos é a mesma para listagem e alteração.

Resta-nos agora a implementação para o botão que cancela o processo de alteração. Veja esta implementação a seguir, que deve ser realizada no arquivo `tipo-servicos-add-edit.page.ts`. Note que atribuímos ao `FormGroup` o objeto inicial, recuperado na construção do componente, cancelando qualquer alteração realizada nos controles e retornando nossa variável de controle ao estado de visualização.

```
cancelarEdicao() {
    this.tiposServicosForm.setValue(this.tipoServico);
    this.modoDeEdicao = false;
}
```

4.6 Navegação para uma página de inserção de dados utilizando classes de negócio

Em uma página onde os dados são listados é muito comum ser possível navegar para uma página de inserção de novos dados para a listagem em foco. Vamos fazer uso de um controle Ionic chamado `FAB` (*Floating Action Button*) para que o usuário possa realizar essa navegação. Vamos inseri-lo em nosso template de listagem, antes do fechamento da tag `<ion-content>`. Veja o código a seguir.

```
<ion-fab vertical="top" horizontal="end" slot="fixed" edge="true">
    <ion-fab-button routerLink='/tipo-servicos-add-edit/-1'>
        <ion-icon name="add"></ion-icon>
    </ion-fab-button>
</ion-fab>
```

Execute sua aplicação e note, na página de listagem, no topo (`top`) à direita (`end`), às margens da listagem e cabeçalho (`edge`), a inserção de um botão em forma de círculo com o ícone de adição (`add`). Notou as diretivas e valores durante a sentença anterior entre parênteses? Observou também o valor negativo enviado na rota? Compare sua visão com a da figura a seguir.

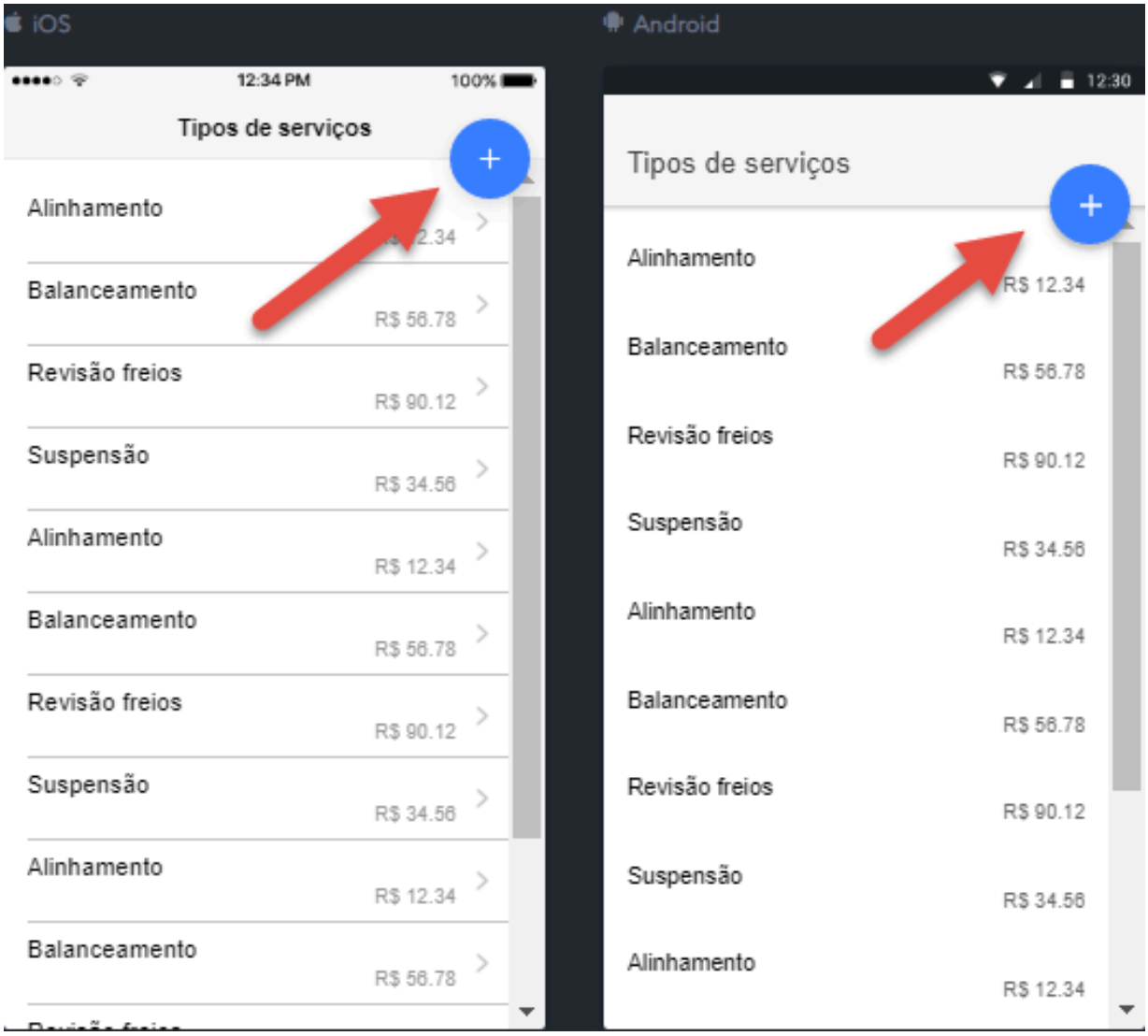


Figura 4.4: FAB para adicionar novos valores.

Vamos começar a implementação da funcionalidade para o FAB trazendo para nosso projeto o uso de classes de modelo. Em TypeScript, quando temos um modelo apenas de propriedades, a recomendação é o uso de `interface` em vez de `class`. Vamos então criar, na pasta `app`, uma nova pasta chamada `models`. Dentro dela, crie um arquivo chamado `tipo-servico.model.ts` e, dentro dele, insira o código a seguir.

```
export interface TipoServico {
    id: number;
    nome: string;
```

```
    valor: number;
}
```

Observe que estamos criando a interface de maneira semelhante à que criamos as classes de nossos componentes, apenas utilizando a palavra reservada `interface`.

A utilização de modelos de negócio, princípio da Orientação a Objetos, nos permitirá tipificar nossos objetos, arrays e coleções, que até então aceitam `any` como tipo de dados. Com essa nova técnica, precisamos tipificar o objeto array em nosso serviço e podemos fazer isso alterando a instrução de declaração da matriz para como está exibido na sequência. Na listagem, também está a nova assinatura para alguns métodos, também da classe de serviço (`tipo-servicos.service.ts`).

```
// Declaração da array
private tiposServicos: TipoServico[] = [
    { id: 1, nome: 'Alinhamento', valor: 12.34 },
    { id: 2, nome: 'Balanceamento', valor: 56.78 },
    { id: 3, nome: 'Revisão freios', valor: 90.12 },
    { id: 4, nome: 'Suspensão', valor: 34.56 }
];

// Assinatura de métodos com tipo de retorno
getById(id: number): TipoServico

getAll(): TipoServico[]

// Tipificação do argumento recebido
update(tipoServico: TipoServico)
```

Precisamos também alterar a declaração de nossa variável que armazenará o tipo de serviço em tela. Veja na sequência. Observe também que já aproveitamos e o tornamos privado, pois não o utilizaremos mais em nosso template, isso em `tipo-servicos-add-edit.page.ts`.

```
private tipoServico: TipoServico;
```

Precisamos pensar no fato de que nosso componente e template são responsáveis pela visualização do item selecionado na listagem, em sua alteração, e agora em inserção de um novo item. Onde temos os dados a serem exibidos na página recuperados? No método `ngOnInit()` do componente (`tipo-servicos-add-edit.page.ts`). Vamos adaptá-lo para o código a seguir.

```
ngOnInit() {
    const id: number = Number(this.route.snapshot.paramMap.get('id'));

    if (id > 0) {
        this.tipoServico = this.tipoServicoService.getById(id);
    } else {
        this.tipoServico = {id, nome: '', valor: 0.00 };
        this.modoDeEdicao = true;
    }

    this.tiposServicosForm = this.formBuilder.group({
        id: [this.tipoServico.id],
        nome: [this.tipoServico.nome, Validators.required],
        valor: [this.tipoServico.valor, Validators.required]
    });
}
```

Veja que, tirando o bloco `if...else` , o resto já conhecemos. A diferença está no momento de inicializarmos a propriedade `tipoServico` , que, até antes do código anterior, era inicializada apenas pela recuperação de um item já existente em nosso serviço. Agora isso mudou, pois temos a situação de envio de um parâmetro com valor `-1` , que sinalizará para nossa aplicação que queremos uma interface de inserção. Por isso, nosso `else` tem as instruções de inicialização da propriedade e alteração de nosso flag de edição, a `modoDeEdicao` .

Mas e quando formos gravar? O id negativo precisará receber um valor válido, que, para nossa lógica simples, será o valor seguinte ao último item de nosso array, e faremos isso no momento da

atualização. Veja, em nossa classe de serviço (`tipo-servicos.service`), o novo código para o `update()` .

```
update(tipoServico: TipoServico) {
    if (tipoServico.id < 0) {
      tipoServico.id = this.tiposServicos[this.tiposServicos.length -
1].id + 1;
      this.tiposServicos.push(tipoServico);
    } else {
      this.tiposServicos[this.getIndexOfElement(tipoServico.id)] =
tipoServico;
    }
}
```

Veja que fazemos também uso de um `if...else` para que o método saiba o que deve fazer com o objeto recebido, ou seja, é preciso inserir ou atualizar e, com base no `id` , o método saberá o que fazer. Atribuímos o valor do id ao novo objeto e então realizamos o `push()` para que ele seja inserido na matriz.

Podemos agora concluir nossa implementação referente à inserção e alteração exibindo ao usuário uma mensagem sobre a realização do processo e um redirecionamento dele para a página de listagem.

No capítulo anterior, vimos algumas possibilidades para exibição de mensagens ao usuário e vamos optar agora pelo `Toaster` . Entretanto, em vez de criarmos em nosso componente um método que exibirá o Toaster, vamos utilizar um recurso aprendido aqui, o *services*.

Na pasta `services` , que já temos na estrutura da nossa aplicação, crie um arquivo chamado `toast.service.ts` e, como código, implemente a listagem a seguir. Veja que o método é o mesmo que implementamos no capítulo anterior.

```
import { ToastController } from '@ionic/angular';
import { Injectable } from '@angular/core';

@Injectable({
```

```
    providedIn: 'root'
  })
export class ToastService {
    constructor(private toastCtrl: ToastController) {}

    async presentToast(message: string, duration: number, position: 'top'
| 'bottom') {
        const toast = await this.toastCtrl.create({
          message,
          duration,
          position
        });
        toast.present();
    }
}
```

Agora precisamos consumir esse serviço em nosso componente de edição/inserção de um tipo de serviço. Assim, no construtor da classe do arquivo `tipo-servicos-add-edit.page.ts`, insira o código a seguir como argumento.

```
private toastService: ToastService
```

Como temos um objeto `ToastService` injetado, podemos efetivamente consumi-lo e faremos isso em nosso método `submit()` após a chamada ao `update()` inserindo o código a seguir.

```
this.toastService.presentToast('Gravação bem sucedida', 3000, 'top');
```

Enfim, para redirecionarmos a aplicação para nossa listagem inicial, precisamos injetar também em nossa classe um objeto responsável por nossas rotas, o que nos leva a inserir o código a seguir em nosso construtor.

```
private router: Router
```

Para efetivar o redirecionamento, após a invocação ao nosso método para exibição do `Toaster`, precisamos implementar a instrução a seguir, que redireciona a aplicação para a rota `''`, que levará para a `home`.

```
this.router.navigateByUrl('');
```

Uma observação: como agora estamos redirecionando a aplicação após a gravação, seja ela inserção ou atualização, não precisamos nos preocupar mais com a variável de controle de estado em edição no método `submit()`. Por outro lado, podemos pensar no redirecionamento apenas quando for uma atualização e, quando estivermos inserindo um registro, a página será disponibilizada para um novo tipo de serviço.

4.7 Remoção de um item da lista

Estamos quase concluindo nossas operações CRUD para tipos de serviços, mas você já sabe o que é CRUD? Caso não saiba, CRUD é um acrônimo para *Create, Read, Update e Delete* (Criação, Leitura, Atualização e Remoção). Isso quer dizer que, quando estamos implementando um CRUD, estamos desenvolvendo um serviço que possibilita essas quatro operações básicas para um determinado tipo de dado.

Até aqui temos a criação (inserção), leitura (busca por todos os registros e pelo id) e a atualização (alteração). Falta apenas a remoção de dados de nossa matriz, e é esse o nosso objetivo nesta seção.

Em listagens nas aplicações móveis, temos um recurso intuitivo que todos os usuários seguem: o de arrastar um item. Isso é conhecido como `slide`, que, traduzindo para nossa funcionalidade, seria `deslizar`, que fica melhor que o "arrastar", usado em aplicações de janelas e que nos remete ao `Drag and Drop`.

Com isso posto, podemos abstrair que precisamos adaptar nossa listagem para possibilitar ao usuário essa funcionalidade de deslizar um item e, a partir desta operação, apresentar uma opção para remover o item em referência. Vamos adaptar nosso código no

arquivo HTML de listagem para o `<ion-list>`. Veja o novo código na sequência.

```
<ion-list #slidingList>
    <ion-item-sliding *ngFor="let tipoServico of tiposServicos">
      <ion-item routerLink='/tipo-servicos-add-edit/{{tipoServico.id}}'>
        <ion-label>
          <h2>{{tipoServico.nome}}</h2>
          <p style="text-align:right">
            R$ {{tipoServico.valor}}</p>
        </ion-label>
      </ion-item>

      <ion-item-options side="end">
        <ion-item-option color="danger"
(click)="removerTipoServico(tipoServico)">
          <ion-button color="danger" no-padding>
              <ion-icon name="trash"></ion-icon>
              Remover
          </ion-button>
        </ion-item-option>
      </ion-item-options>
    </ion-item-sliding>
</ion-list>
```

Verifique, no início do conteúdo de `<ion-list>`, o uso de um novo componente, o `<ion-item-sliding>`. Esse componente tem em si a seleção dos itens que serão iterados, tirando essa funcionalidade do `<ion-item>`. Ainda nesta tag, veja a variável de template `#slidingList`. Ela terá a finalidade de um *workaround* para que, ao removermos um item, seja possível deslizarmos outro.

Após o `<ion-item>`, temos um componente responsável pelo comportamento que desejamos quando o usuário deslizar um item da listagem. Estamos falando do `<ion-item-options>`, que define um conjunto de `<ion-item-option>` para as possíveis opções a serem oferecidas ao usuário após o deslizamento do item. Em nosso caso, teremos apenas a de remoção do item da coleção que popula a

listagem, o que nos leva a um único `<ion-item-option>`, que tem nele a definição de um `<ion-button>` com um `<ion-icon>`.

Observe que no `<ion-item-option>` temos um *event binding* com `removerTipoServico()`, que receberá o serviço atual, exibido no item deslizado da listagem. Logo implementaremos esse método.

Na tag `<ion-item-options>`, observe a definição do atributo `side` com `"end"`, o que significa que as opções para cada item estarão exibidas ao lado direito do item deslizado. Outra opção seria `start`, que os colocaria ao lado esquerdo.

Já para a exibição do ícone tradicional de lixeira no botão, estamos dando ao `<ion-icon>`, na propriedade `name`, o valor `trash`, o que refletirá no ícone desejado. Esses nomes e figuras podem ser localizados em https://ionicframework.com/docs/ionicons/. Veja a figura a seguir com a exibição dessas opções.

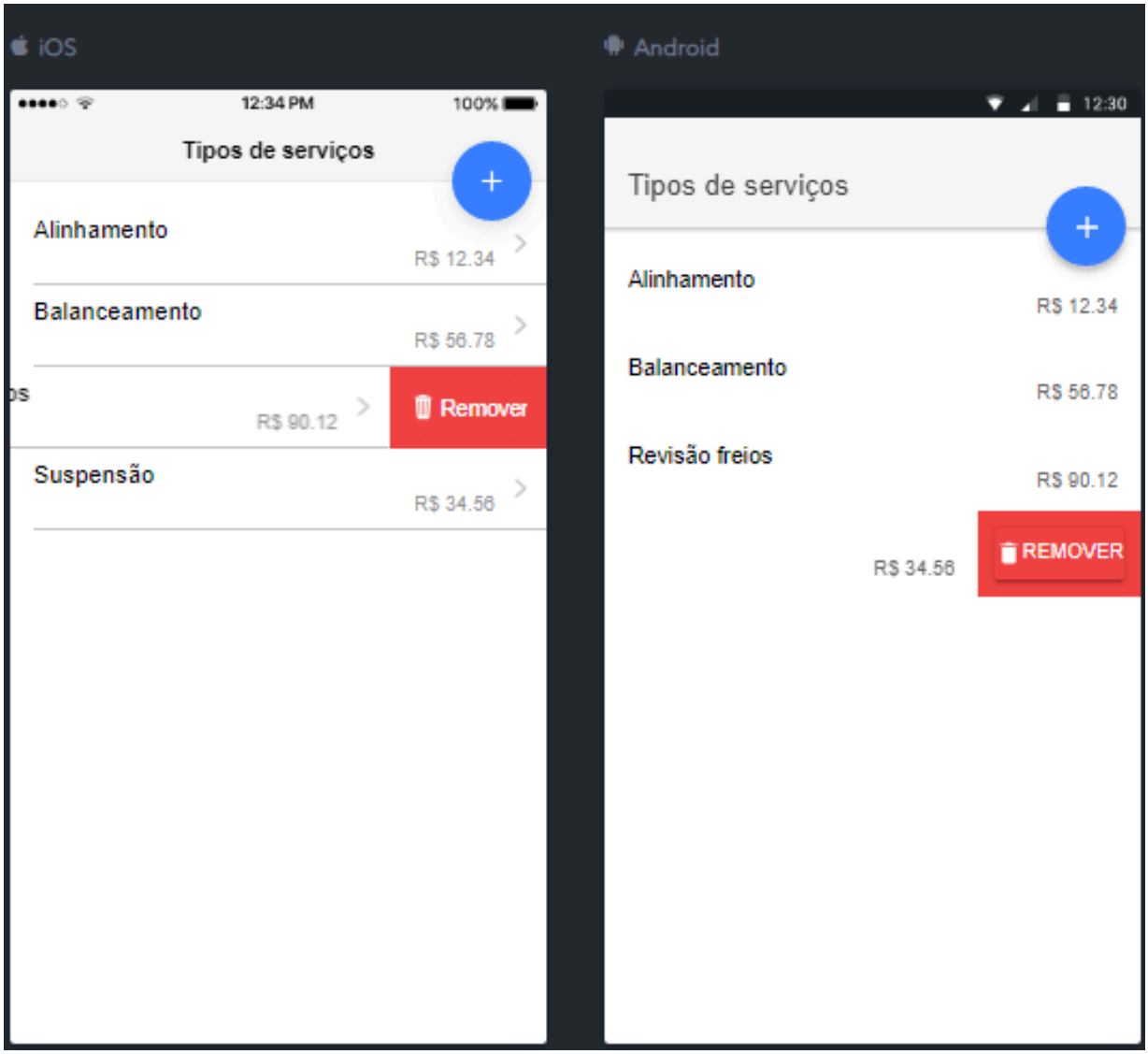


Figura 4.5: Deslizando um item da listagem.

Precisamos agora implementar nosso componente para que as implementações que fizemos no template possam ter seu comportamento capturado e executado de acordo com o esperado. Nosso primeiro passo é recuperar nossa variável de template `slidingList`, que definimos no código anterior. Para isso, antes do construtor, insira o código a seguir.

```
@ViewChild('slidingList') slidingList: IonList;
```

Precisamos criar o nosso método `removerTipoServico()`, que será invocado quando o usuário confirmar o desejo de remoção do item deslizado. Veja o código para esse método na listagem a seguir. Observe a definição do método como assíncrono, pois nossa última instrução é `await` e é ela que permitirá que após a remoção de um item, outros deslizes possam ser realizados.

```
async removerTipoServico(tipoServico: TipoServico) {
    this.tipoServicoService.remove(tipoServico);
    this.toastService.presentToast('Tipo de serviço removido', 3000,
'top');
    await this.slidingList.closeSlidingItems();
}
```

A implementação do método anterior depende de outras antes de podermos testar nossa aplicação. A primeira refere-se a implementação do método `remove()` em nosso `TipoServicosService`. Veja-o na sequência.

```
remove(tipoServico: TipoServico) {
    this.tiposServicos.splice(this.getIndexOfElement(tipoServico.id), 1);
}
```

A segunda alteração que precisamos realizar é em relação ao `this.toastService`, que precisamos também injetar em nosso construtor, tal qual fizemos para a página de inserção/consulta/alteração. Veja o código a seguir, que devemos inserir em nosso construtor na classe de listagem de tipos de serviço.

```
private toastService: ToastService
```

Bora testar?

Conclusão

Terminamos este interessante capítulo, onde realizamos toda a operação relacionada a um CRUD. Vimos vários recursos e técnicas,

que poderemos utilizar em nossas aplicações futuras e certamente usaremos nos capítulos seguintes.

Recapitulando e relembrando, vimos novamente o processo de criação de um projeto, criamos uma página de listagem, populando-a com itens armazenados em uma matriz, fizemos uso de `Services`, aprendemos e praticamos a navegação entre páginas, enviando objetos para definição de comportamento após a navegação.

Foi possível inserirmos um item em nossa matriz, alterá-lo e removê-lo, mantendo sempre nossos controles visuais atualizados e com comportamentos de acordo com seu estado.

É hora de tomarmos um copo de água e partirmos para o próximo capítulo, no qual veremos um pouco de persistência local.

Capítulo 5
Persistindo dados fisicamente em aplicações Ionic

No capítulo anterior, implementamos um CRUD para o modelo de negócio `Tipo de Serviço`, onde os dados eram mantidos em uma matriz, o que não garantia sua persistência entre execuções da aplicação.

Veremos neste quinto capítulo alguns mecanismos para persistência física de dados. Aplicaremos estes recursos e técnicas em modelos de negócio relacionados a peças a serem utilizadas em manutenções de veículos, com um CRUD completo, e começaremos também um CRUD com ordem de serviço registrada na entrada do veículo. Trabalharemos o armazenamento interno do Ionic, conhecido com Storage, para peças, e depois com ordens de serviço trabalharemos a metodologia mais comum, uma base de dados, com tabelas.

5.1 Criação dos artefatos de modelo e de registro

Vamos dar continuidade ao projeto que trabalhamos no capítulo anterior. Vamos implementar nosso modelo de negócio e depois nossa página de registro de dados, onde persistiremos fisicamente nossos objetos de peças de veículo.

Vamos conversar sobre o `id` que será atribuído a cada objeto. Precisamos garantir que cada valor seja único, o que nos leva a pensar em um valor do tipo GUID, ou UUID. Caso você ainda não conheça este tipo de dado, recomendo uma leitura básica em https://pt.wikipedia.org/wiki/Identificador_%C3%BAnico_universal, que você pode fazer quando julgar interessante. Mas, em resumo,

tipos de dados GUID garantem sempre que seu valor NUNCA se repetirá.

Existe um componente no NPM que agrega a nossas aplicações a possibilidade de termos este tipo de dado disponível, o `guid-typescript`. Precisamos instalá-lo e faremos isso acessando o terminal na pasta de nossa aplicação. Estando lá, digite a seguinte instrução no prompt. Atente à versão que estamos utilizando.

```
npm i guid-typescript
```

Com nosso componente instalado na pasta `src/app`, se você criou um novo projeto, crie uma pasta chamada `models` e, dentro dela, o arquivo `peca.model.ts`, tal qual vemos no código a seguir. Caso esteja seguindo o projeto anterior, a pasta já existe. Esta situação pode se repetir na sequência, ok? Observe a importação do componente instalado.

```
import { Guid } from 'guid-typescript';

export interface Peca {
    id: Guid;
    nome: string;
    valor: number;
}
```

Vamos criar os recursos necessários para trabalharmos com o processo de inserção e edição de peças. Tal qual estamos fazendo, com o auxílio do `Ionic generate`. Desta maneira, no terminal, na pasta do projeto, execute a instrução a seguir e, após a criação dos arquivos, vamos à adequação deles.

```
ionic generate page pages/pecas/pecas-add-edit
```

pecas-add-edit.page.html

Observe no código a seguir o caminho para a figura que utilizamos e lembre-se de que a pasta `assets` está no mesmo nível de `app` e você precisará criar a `imgs` e copiar a figura para esta pasta. Já

aproveitamos neste momento para deixar o template preparado para a edição e inserção de peças, tal qual fizemos para `Tipos de Serviços`.

```
<ion-header>
    <ion-toolbar>
        <ion-buttons slot="start">
            <ion-back-button defaultHref=""></ion-back-button>
        </ion-buttons>
        <ion-title>Peças</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content padding>
    <ion-item-divider color="light">
        <ion-grid no-padding>
            <ion-row no-padding>
                <ion-col col-3 align-self-center>
                    <ion-img style="width: 40px; height: 40px;"
src="assets/imgs/tab_pecas.png"></ion-img>
                </ion-col>
                <ion-col col-9>
                    <ion-label style="font-size: 24px">Dados da peça</ion-
label>
                </ion-col>
            </ion-row>
        </ion-grid>
    </ion-item-divider>

    <form [formGroup]="pecasForm" (submit)="submit()">
        <ion-item lines="none" [disabled]="!modoDeEdicao">
            <ion-label position="floating" class="label-input-
title">Nome</ion-label>
            <ion-input formControlName="nome" type="text" style="margin-
left: 3%;"></ion-input>
        </ion-item>

        <ion-item lines="none" [disabled]="!modoDeEdicao">
            <ion-label position="floating" class="label-input-
```

```
title">Valor</ion-label>
            <ion-input text-right formControlName="valor" type="number">
</ion-input>
        </ion-item>

        <ion-button *ngIf="!modoDeEdicao" shape="round" color="primary"
expand="block" padding (click)="iniciarEdicao()">
            Alterar dados
        </ion-button>
        <div *ngIf="modoDeEdicao" no-padding>
            <ion-grid>
                <ion-row>
                    <ion-col col-6 >
                        <ion-button shape="round" color="success"
size="small" padding type="submit" expand="block">
                            Gravar
                        </ion-button>
                    </ion-col>

                    <ion-col col-6>
                        <ion-button shape="round" color="warning"
size="small" padding (click)="cancelarEdicao()"  expand="block">
                            Cancelar
                        </ion-button>
                    </ion-col>
                </ion-row>
            </ion-grid>
        </div>
    </form>
</ion-content>
```

pecas-add-edit.page.ts

Observe que no código a seguir já temos a implementação praticamente concluída, restando-nos apenas pontos específicos deste capítulo, relacionados à persistência, e que veremos de maneira dedicada logo na sequência. Veja que também fazemos uso de nosso `Toast Service`, que já temos implementado do capítulo anterior e podemos reutilizá-lo. Temos no código alguns comentários

do que precisamos ainda implementar. Verifique o uso do componente `Guid` no método `OnInit()`.

```
import { Component, OnInit } from '@angular/core';
import { Peca } from 'src/app/models/peca.model';
import { FormGroup, FormBuilder, Validators } from '@angular/forms';
import { ActivatedRoute } from '@angular/router';
import { Guid } from 'guid-typescript';
import { ToastService } from 'src/app/services/toast.service';

@Component({
    templateUrl: './pecas-add-edit.page.html'
})
export class PecasAddEditPage implements OnInit {
    private peca: Peca;
    public modoDeEdicao = false;
    public pecasForm: FormGroup;

    constructor(
        private formBuilder: FormBuilder,
        private route: ActivatedRoute,
        private toastService: ToastService,
    ) {}

    iniciarEdicao() {
        this.modoDeEdicao = true;
    }

    cancelarEdicao() {
        this.pecasForm.setValue(this.peca);
        this.modoDeEdicao = false;
    }

    async submit() {
        // Atualizar/Inserir objeto de maneira persistida
        this.toastService.presentToast('Gravação bem sucedida', 3000,
'top');
        this.modoDeEdicao = false;
        // Navegar para a página principal
```

```
    }

    ngOnInit() {
        const id = this.route.snapshot.paramMap.get('id');

        if (id && Guid.isGuid(id)) {
            // Recuperaremos o objeto persistido
        } else {
            this.peca = {id: Guid.createEmpty(), nome: '', valor: 0.00 };
            this.modoDeEdicao = true;
        }

        this.pecasForm = this.formBuilder.group({
            id: [this.peca.id],
            nome: [this.peca.nome, Validators.required],
            valor: [this.peca.valor, Validators.required]
        });
    }
}
```

Em relação ao `pecas-add-edit-routing.module.ts` não precisamos ajustar nada por enquanto. Já para o `pecas-add-edit.module.ts`, precisamos inserir `ReactiveFormsModule`, aos `imports` de `@NgModule`. Também precisaremos inserir, em `providers` de `app.module.ts` o `FormBuilder`. Para facilitar, trago este último código na sequência.

```
providers: [
    FormBuilder, { provide: RouteReuseStrategy, useClass:
IonicRouteStrategy }],
```

Precisamos alterar a rota inicial de nossa aplicação para a página que acabamos de implementar e devemos fazer isso no arquivo `app-routing.module.ts`, trocando em `redirectTo`, do `path> ''`, o valor que lá está por `pecas-add-edit`.

Com esses códigos implementados, já podemos testar nossa aplicação. Veja a figura a seguir com a exibição da página para inserção de uma nova peça.

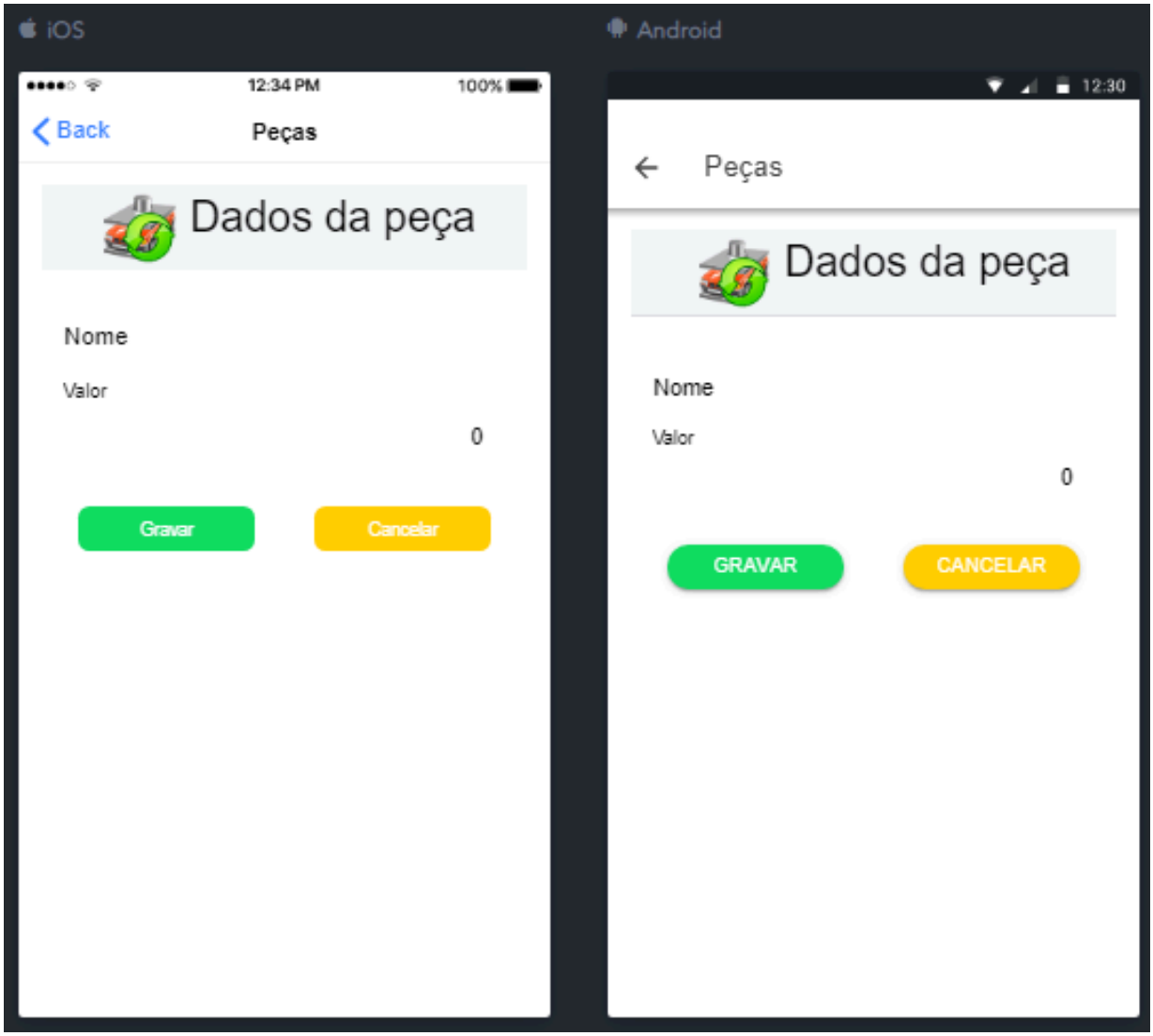


Figura 5.1: Página de inserção e alteração de peças

5.2 Persistência no Ionic com Storage para a inserção de dados

O Ionic possui um mecanismo de persistência nativo, ligado a uma API chamada `Storage`, que faz uso de grande variedade de

mecanismos de persistência, dentre os quais você pode escolher o melhor, em sua análise, para as plataformas que estiver desenvolvendo.

Inicialmente não especificaremos nenhum mecanismo de persistência, deixaremos que o Ionic utilize seu mecanismo nativo, que é o `IndexedDB`, disponibilizado no lado cliente, que em nosso caso é a engine do navegador.

Para implementarmos nosso primeiro código relacionado à inserção de uma peça, em uma base de dados do `IndexedDB`, precisamos instalar um plugin em nosso projeto, e isso pode ser feito no terminal, na pasta dele, com a instrução a seguir.

```
npm install @ionic/storage-angular
```

Uma vez que o componente esteja instalado em nossa aplicação, precisamos disponibilizá-lo para nosso componente e, para isso, inseriremos as seguintes instruções.

```
// Junto aos demais imports de app.module.ts
import { IonicStorageModule } from '@ionic/storage-angular';

// E, ao final dos imports dentro do NgModule de app.module.ts
, IonicStorageModule.forRoot()
```

Agora que temos nosso mecanismo para persistência disponibilizado para nossa aplicação, precisamos implementar um serviço que atenderá às interações que exijam busca e armazenamento de dados na base. Na pasta `services`, crie um arquivo chamado `pecas.service.ts`, com o código a seguir.

```
import { Injectable } from '@angular/core';
import { Peca } from '../models/peca.model';
import { Guid } from 'guid-typescript';
import { Storage } from '@ionic/storage-angular';

@Injectable({
  providedIn: 'root'
})
```

```
export class PecasService {

  constructor(
    private storage: Storage
  ) { }

  async update(peca: Peca) {
    await this.storage.create();
    if (peca.id.isEmpty()) {
      peca.id = Guid.create();
    }

    this.storage.set(peca.id.toString(), JSON.stringify(peca));
  }
}
```

Veja que importamos `Storage` e o recebemos por injeção pelo construtor. Temos definido, inicialmente, o método para atualizar a base de dados. Dê uma lida na última instrução, que invoca o `set()` do objeto `storage`. Consegue identificar que estamos armazenando um par `chave/valor`? Pois bem, é assim que o `IndexedDB` trabalha, e a chave precisa ser uma string. A lógica do método `update()` é simples: caso seja um novo objeto, sem `id`, este valor será então gerado. Após essa verificação, o objeto é atribuído à nossa base.

Importante você verificar a primeira instrução do método, que é a responsável pela criação da base de dados no `IndexedDB`. Este não é o melhor local para este procedimento, mas no momento, deixaremos aqui mesmo.

Precisamos injetar nosso serviço no construtor. Sei que já fizemos isso, mas para reforçar, veja as instruções a seguir.

```
// Junto com demais imports do componente pecas-add-edit.page.ts
import { PecasService } from 'src/app/services/pecas.service';

// Código a ser inserido no construtor
private pecasService: PecasService
```

Vamos fechar esta etapa digitando as instruções a seguir logo no início do método `submit()`. Precisamos invocar o `update()`, enviando o formulário completo para atualização e, na segunda instrução, atribuindo a `this.peca` o novo objeto, agora gravado. Insira o `async` antes do nome do método `update()`, caso você ainda não o tenha feito.

```
await this.pecasService.update(this.pecasForm.value);
this.peca = this.pecasForm.value;
```

Teste sua aplicação no navegador, informe o nome da peça e seu respectivo valor. Clique no botão gravar. Vamos ver como os dados foram persistidos?

Utilizando o Chrome, acesse as ferramentas do desenvolvedor (F12) e, em `Application`, localize a base de dados e veja no painel os dados que foram armazenados. Consegue se localizar na figura a seguir? O que achou?

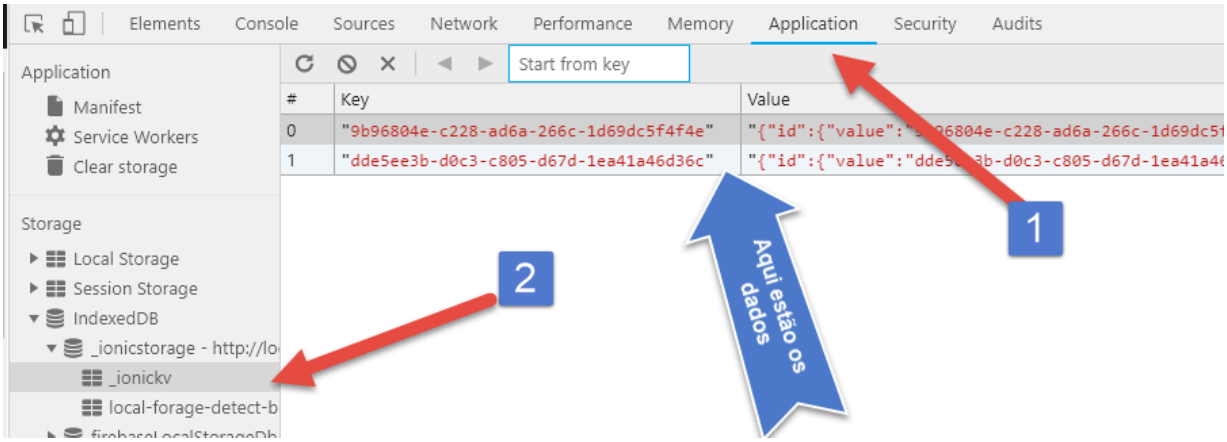


Figura 5.2: Verificação dos dados persistidos no IndexedDB

Caso seus dados não apareçam como dito anteriormente, procure no `WebSQL`.

5.3 Recuperação de dados persistidos com Storage

No capítulo anterior, começamos nossa aplicação com a página de listagem de dados e, a partir dela, navegávamos para uma operação de consulta, alteração, remoção ou inclusão. Neste capítulo, a escolha foi começar com a inserção, pois, como os dados são persistidos fisicamente, precisamos deles para implementar as demais funcionalidades.

Como já temos os dados persistidos, pois realizamos isso quando concluímos a seção anterior, podemos começar a trabalhar com a página de listagem dos elementos armazenados e faremos isso com a implementação de um método em nossa classe de serviços para peças, a `pecas.service.ts`. Veja este código na sequência. Observe a primeira instrução do método, que parece um comentário, mas é uma diretiva para o TypeScript ignorar a indicação por preferência ao uso de constantes.

```
async getAll() {
    // tslint:disable-next-line:prefer-const
    let pecas: Peca[] = [];
    try {
      await this.storage.forEach((value: string, key: string) => {
        const peca: Peca = JSON.parse(value);
        pecas.push(peca);
      });
      return pecas;
    } catch (error) {
      return error;
    }
}
```

No código anterior, declaramos um array, que será populado com os objetos recuperados do armazenamento físico por meio da invocação assíncrona ao método `forEach()`. Uma vez que todos os

objetos estejam recuperados e não haja erro durante o processo de recuperação, a matriz é retornada ao chamador.

Resta-nos implementar nossa página de listagem de peças.Veja a seguir a instrução a ser executada no terminal e na sequência as orientações para ajustes nos arquivos.

```
ionic generate page pages/pecas/pecas-listagem
```

pecas-listagem.page.html

Se verificar no `routerLink` para o primeiro `<ion-item>`, notará que o binding está ligado a um método, o `idAsString()`. Precisamos deste artifício para que o parâmetro possa ser enviado corretamente, como `String`, uma vez que o temos armazenado como `Guid`. Isso poderia ser contornado se criássemos uma propriedade do tipo `String` em vez de `Guid` e, na hora de armazenarmos o valor nela, o convertêssemos. Ainda, poderíamos utilizar a definição do modelo em uma classe e não interface e, então, definir na classe um método que resolvesse a conversão na obtenção do dado. Aqui optei por essa técnica para que possamos ver didaticamente a conversão de um dado no registro de links para rotas.

```
<ion-header>
    <ion-toolbar>
      <ion-title>Peças</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content padding-left padding-top>
  <ion-list #slidingList>
    <ion-item-sliding *ngFor="let peca of pecas">
      <ion-item routerLink='/pecas-add-edit/{{idAsString(peca.id)}}'>
        <ion-label>
          <h2>{{peca.nome}}</h2>
          <p style="text-align:right">R$ {{peca.valor}}</p>
        </ion-label>
      </ion-item>
```

```
    </ion-item-sliding>
  </ion-list>

  <ion-fab vertical="top" horizontal="end" slot="fixed" edge="true">
    <ion-fab-button routerLink='/pecas-add-edit/-1'>
      <ion-icon name="add"></ion-icon>
    </ion-fab-button>
  </ion-fab>
</ion-content>
```

pecas-listagem.page.ts

Nosso `ngOnInit()` traz o uso de uma `Promise` para popular nossa coleção que será utilizada pelo template. Isso é necessário pelo fato de termos o `getAll()` definido como assíncrono. Poderíamos optar por transformar o `ngOnInit()` em assíncrono e realizar a invocação com `await`, em vez de trabalhar o `Promise`. Note também a implementação do método `idAsString()` para converter o `id` de `Guid` para `String` e em seguida para formato JSON.

Um detalhe: a documentação da biblioteca `Guid` aponta que, para retornar uma string de um valor Guid, bastaria invocar o método `toString()` da instância desejada, mas isso não funcionou aqui, mas funcionou na inserção, lembra? Na realidade, quando `id` chega, ao ser verificado, vem como `object Object`, o que impediu certamente a execução deste processo, por isso o *workaround* utilizado no método.

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { PecasService } from '../../../services/pecas.service';
import { Peca } from '../../../models/peca.model';
import { ToastService } from '../../../services/toast.service';
import { IonList } from '@ionic/angular';
import { Guid } from 'guid-typescript';

@Component({
    templateUrl: './pecas-listagem.page.html'
})
```

```
export class PecasListagemPage implements OnInit {

    public pecas;
    @ViewChild('slidingList') slidingList: IonList;

    constructor(
        private pecasService: PecasService,
        private toastService: ToastService) {
    }

    idAsString(id: Guid): string {
        const convertedId = JSON.parse(JSON.stringify(id));
        return convertedId.value;
    }

    ngOnInit(): void {
        this.pecasService.getAll().then(pecas => {
            this.pecas = pecas;
        });
    }
}
```

Em relação aos arquivos `pecas-listagem-routing.module.ts` e `pecas-listagem.module.ts` não há necessidade de alterações neste momento.

Com as implementações anteriores realizadas, precisamos alterar nossa rota `home`. Vamos novamente copiar `pecas-listagem` para o `redirectTo` de `path: ''` e ajusta o path `pecas-add-edit` para `pecas-add-edit/:id`.

Antes de executarmos nossa aplicação, vamos alterar a instrução que cria a comunicação com o recurso de armazenamento, que implementamos anteriormente no `update()` de nosso serviço, pois precisamos ter esta conexão em toda a aplicação e não é produtivo e tampouco correto, a cada interação com o banco, executarmos esta instrução.

No `app.component.ts`, em `src/app`, insira as instruções a seguir.

```
// No início do arquivo
import { Storage } from '@ionic/storage-angular';

// Registrar no construtor
private storage: Storage,

// Após o construtor
async ngOnInit() {
    await this.storage.create();
}
```

Na sequência, retire a invocação ao create que temos no início do método `update` de `pecas.service.ts` e então teste sua aplicação. A figura a seguir traz a aplicação em execução em nossos pseudoemuladores.

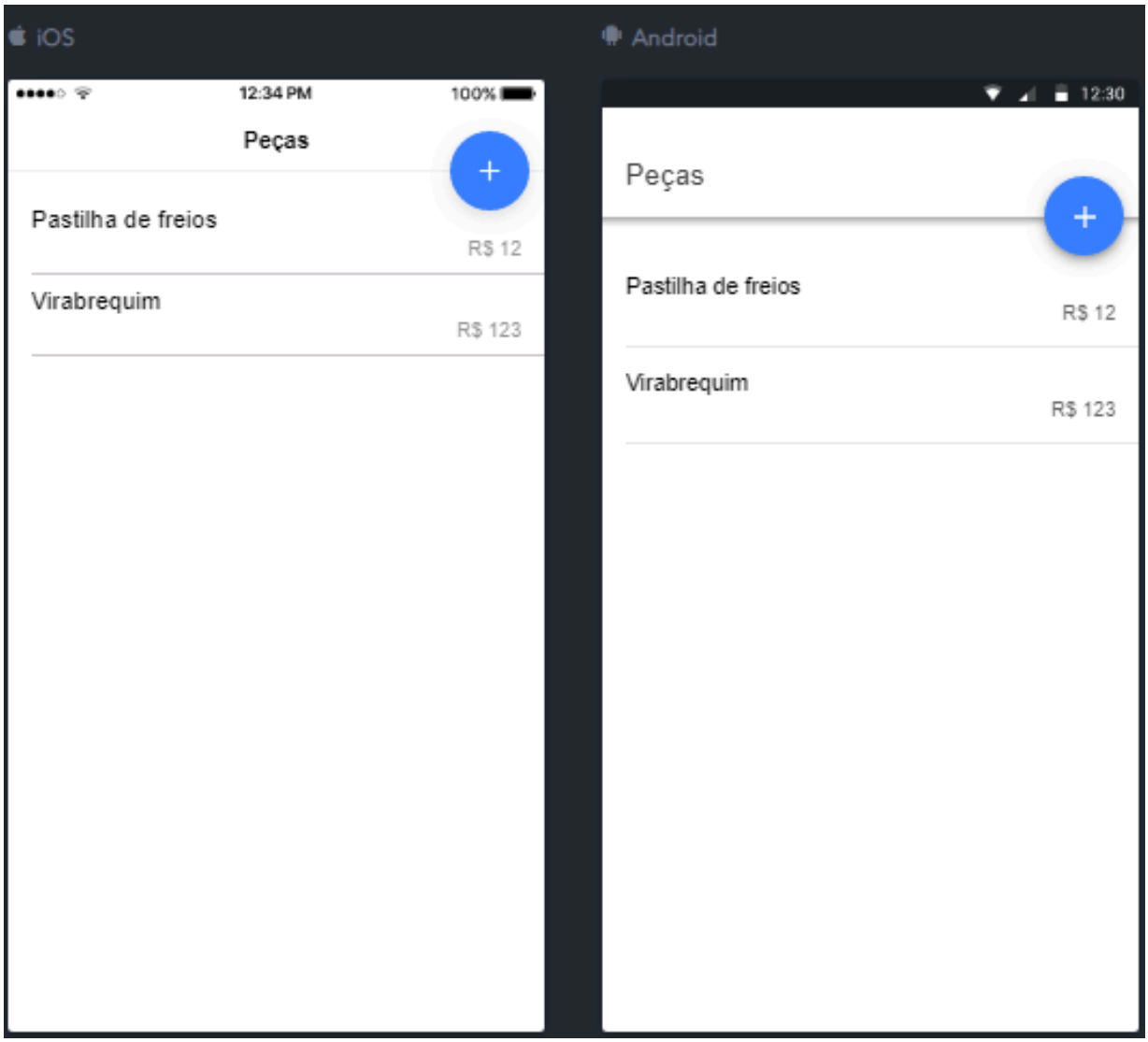


Figura 5.3: Listagem dos dados persistidos no IndexedDB

5.4 Recuperação de uma peça selecionada na listagem

No capítulo anterior, ao pressionarmos um item na página de listagem, éramos direcionados para a página de exibição e alteração dos dados. Precisamos realizar este procedimento aqui também. Teremos que recuperar os dados de nossa persistência física e não

mais de um array. Vamos implementar o método em nossa classe de serviços para peças, a `pecas.service.ts`, assim:

```
async getById(id: string): Promise<Peca> {
    const pecaString = await this.storage.get(id);
    return JSON.parse(pecaString);
}
```

Veja que optamos por definir o tipo de dado de retorno aqui. Quando implementamos o `getAll()`, optamos por não definir de maneira explícita. Como instruções do método, temos a recuperação do valor na base, pelo `id` recebido, e então, com a peça já recuperada, a transformamos de `String` para um objeto JSON.

Vamos mudar um pouco nosso método `ngOnInit` de nosso arquivo `pecas-add-edit.page.ts` para podermos receber o `id` que virá da listagem quando um item for selecionado. O primeiro passo é informarmos que ele será assíncrono, inserindo `async` antes do nome do método. Depois, no `if()` que verifica se recebemos um `id` e se ele é um `Guid`, substituímos o comentário que está lá pelo código a seguir.

```
this.peca = await this.pecasService.getById(id);
```

Com isso temos as alterações necessárias quase finalizadas. Se formos testar nossa aplicação agora, veremos que, ao selecionarmos uma peça, a navegação ocorre, mas não vemos os dados da peça na página de alteração. Isso ocorre por um erro na criação do formulário de validadores (nosso `FormGroup`), que será invocada antes de termos recuperado a peça da base. Precisamos dizer ao nosso template que renderize nosso formulário apenas quando nosso `FormGroup` estiver devidamente criado. Para isso, envolvemos nossas tags `<form></form>` com as tags a seguir.

```
<ng-container *ngIf="pecasForm">
</ng-container>
```

Agora sim. Veja-a em execução na figura a seguir, que traz os dados de uma peça selecionada na página de listagem.

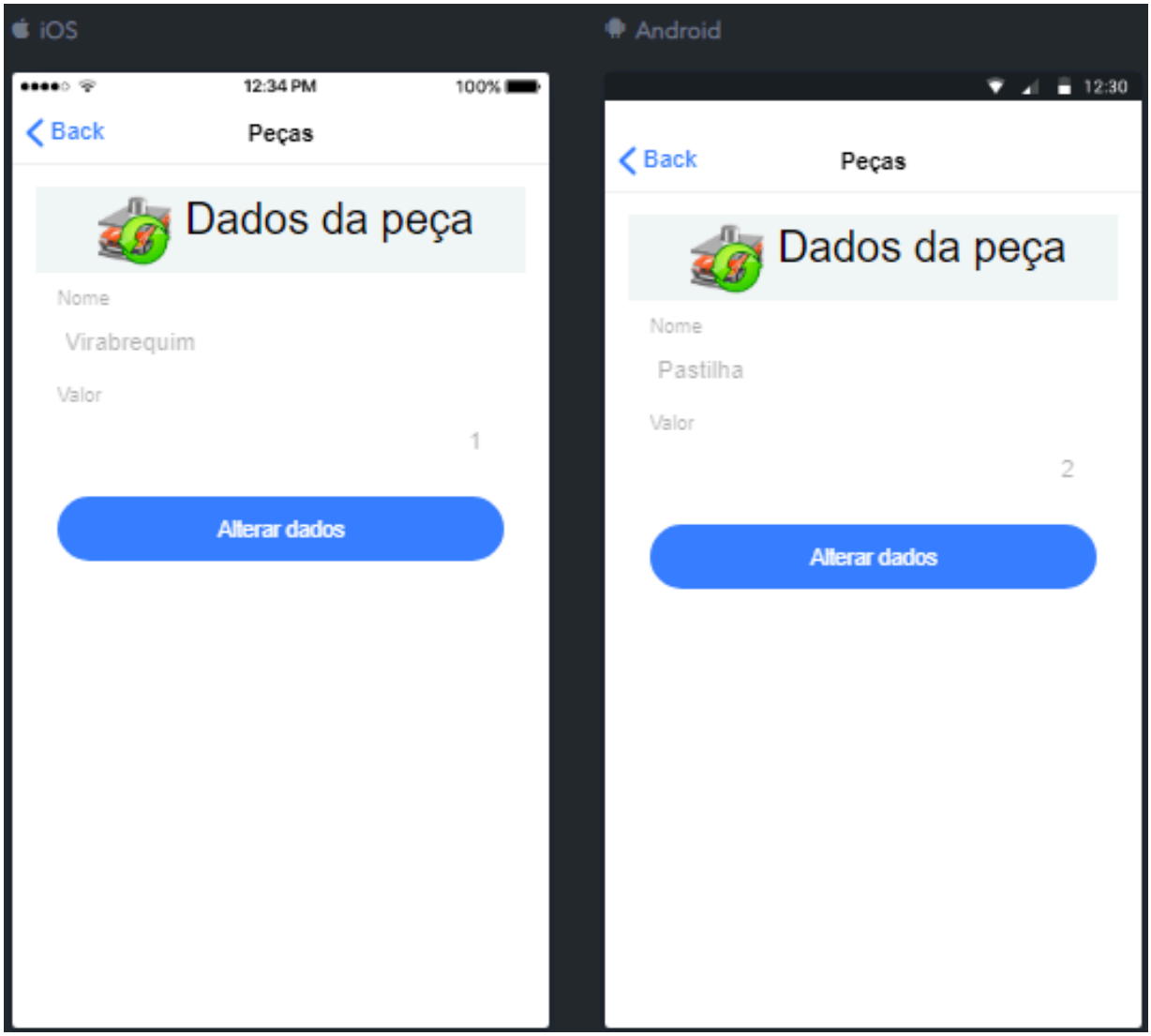


Figura 5.4: Visualização de um dado recuperado da base

5.5 Implementação e validação das operações CRUD

Vamos trabalhar na finalização do CRUD com a adoção da página de listagem dos dados. Já temos a visualização, vamos às demais.

Alteração nos dados de uma peça

Se você, ao executar a aplicação na última solicitação, clicou no botão para permitir a alteração dos dados e depois em gravar, viu que nada ocorreu, pois há um erro de execução, que é exibido na console do navegador e os dados não são alterados. O erro ocorre na invocação do método `isEmpty()` da propriedade `id`, em nossa classe `PecasService`, no método `update()`.

Nosso `submit()` envia os dados do formulário para `update()` e, na construção do formulário, a propriedade `id` da peça em vista perde a referência para o tipo `Guid`, desconhecendo o método `isEmpty()`. Vamos realizar um *workaround* para essa situação. Vamos inserir a instrução a seguir como início do método `update()`, em nosso `pecas.service.ts`. Depois, teste novamente sua aplicação.

```
peca.id = Guid.parse(JSON.parse(JSON.stringify(peca.id)).value);
```

Remoção de uma peça da base de dados

Nossa implementação aqui refere-se à remoção de um item da persistência física e você verá que isso é bem simples. Precisamos inserir a exibição da opção de remover para o usuário. Faremos isso pelo código a seguir, no template, antes de fechar o `<ion-item-sliding>`, no arquivo `pecas-listagem.page,html`.

```
<ion-item-options side="end">
    <ion-item-option color="danger" (click)="removerPeca(peca)">
      <ion-button color="danger" no-padding>
          <ion-icon name="trash"></ion-icon>
          Remover
      </ion-button>
    </ion-item-option>
</ion-item-options>
```

Temos que implementar um método em `pecas.service.ts` e outro em `pecas-listagem.page.ts` e eles estão na sequência.

```
// pecas.service.ts
async removeById(id: string) {
    await this.storage.remove(id);
```

```
}

// pecas-listagem.page.ts
async removerPeca(peca: Peca) {
    await this.pecasService.removeById(this.idAsString(peca.id));
    this.pecas = await this.pecasService.getAll();
    this.toastService.presentToast('Peça removida', 3000, 'top');
    await this.slidingList.closeSlidingItems();
}
```

No segundo método, observe que enviamos para o primeiro uma string que representa o `id` da peça selecionada. Após a remoção, repopulamos nosso objeto utilizado na renderização da página. Terminamos com a exibição de mensagem de sucesso e podemos testar a funcionalidade.

Inserção por meio do FAB Button

No início do capítulo, realizamos o processo de inserção por meio do acesso direto à página com esta funcionalidade e já temos o `<ion-fab-button>` implementado, precisamos adaptar nosso método `submit` para promover a navegação para nossa página de listagem após nossa gravação. Da maneira como estamos fazendo, tanto faz a operação que o usuário esteja realizando, alteração ou inclusão, ao final, ele será redirecionado para a listagem. Veja a seguir o código, que deve ser implementado como última instrução do `submit()`, em `pecas-add-edit.page.ts`, antecedido por uma instrução a ser inserida no construtor.

```
// Inserir no construtor. Atenção ao import
private router: Router

// Última instrução do submit()
this.router.navigateByUrl('');
```

O que acha de agora testarmos todas as funcionalidades de nossa aplicação? Notou que a peça inserida não aparece em nossa

listagem? Pois é. Isso está relacionado ao ciclo de vida de uma visão, nossa página.

Temos em `pecas-listagem.page.ts` a implementação do método `ngOnInit()`, que é disparado quando a página é inicializada da primeira vez. É neste método que temos a população de nosso objeto utilizado para renderização da listagem.

Como dito, este evento ocorre só na primeira vez em que a página é criada. Quando navegamos da inserção ou alteração para a listagem, a página já existe. Precisamos implementar a lógica do `ngOnInit()` em outro método, de outro evento. Você pode optar agora por apagar o comportamento deste método, deixando-o vazio, e então implementar o método da listagem a seguir, no arquivo `pecas-listagem-page.ts`.

```
ionViewWillEnter() {
    this.pecasService.getAll().then(pecas => {
        this.pecas = pecas;
    });
}
```

Veja na listagem anterior o prefixo `ion`, que deixa claro que este método captura um evento do ciclo de vida de uma página/visão, mas agora estamos trabalhando eventos do Ionic e não do Angular. Vamos testar novamente nossa aplicação e ver se o problema foi corrigido?

5.6 SQLite como mecanismo de persistência

Na seção anterior, realizamos operações de CRUD tendo como base o Storage do Ionic. Vimos que os dados ficam atrelados ao navegador e que o acesso a eles se dá por meio de conjuntos de chave/valor. Não fizemos nada grande ou complexo, mas é

importante saber que este mecanismo não é o ideal para uma persistência de dados ao estilo de uma base de dados.

Felizmente temos alguns bancos de dados que funcionam bem em dispositivos móveis, com características de tabelas e consultas. Dentre eles, o mais popular e bem aceito é o SQLite (https://www.sqlite.org/index.html). Nós o utilizaremos em nossa aplicação por meio de um plugin do Capacitor, executando as instruções a seguir.

```
npm install --save @capacitor-community/sqlite
```

Vamos agora partir para a implementação do serviço que nos proverá recursos relacionados à base de dados SQLite que utilizaremos em nossa aplicação. Para isso, na pasta `src/app/services`, vamos criar um arquivo chamado `database.service.ts`, com o código inicial igual ao apresentado na sequência. Verifique a declaração de uma varíavel (`sqlite`), para nosso plugin, uma que nos remeterá à plataforma de execução do aplicativo (`platform`), outra que que nos informará se a execução é em um ambiente nativo ou web (`isNative`) e por fim, uma que nos dará acesso direto ao plugin (`sqlitePlugin`), que seu tipo é `any` mesmo, de acordo à documentação. A ideia de verificação sobre ser ou não nativo é apenas informativo, pois trabalharemos apenas o nativo.

```
import { Injectable } from '@angular/core';
import { SQLiteConnection } from '@capacitor-community/sqlite';

@Injectable({
    providedIn: 'root'
})
export class DatabaseService {
    sqliteConnection: SQLiteConnection;
    platform: string;
    native: boolean = false;
    sqlitePlugin: any;
}
```

Nosso primeiro método a ser implementado será o responsável pela inicialização do plugin no contexto de nossa aplicação. Veja o código a seguir e observe que ele é assíncrono (`Promise`) e retornará um valor booleano. Novas importações serão necessárias e estão antecedendo o método, mas lembre-se que elas são no início do arquivo recém-criado. Alguns comentários após o código, mas fique atento(a) aos imports que serão necessários.

```
async initializePlugin(): Promise<boolean> {
    return new Promise(resolve => {
        this.platform = Capacitor.getPlatform();
        if (this.platform === 'ios' || this.platform === 'android')
this.native = true;
        this.sqlitePlugin = CapacitorSQLite;
        this.sqliteConnection = new SQLiteConnection(this.sqlitePlugin);
        resolve(true);
    });
}
```

O método é praticamente declarativo, buscando dados, como a plataforma, na primeira instrução, depois verificando se o acesso é nativo ou não, buscando o plugin do SQL para o Capacitor, com ele buscando uma conexão com banco e por fim, resolvendo a `Promise` , retornando `true` .

Seguindo o fluxo de criação que estamos fazendo, precisamos agora criar uma conexão com nossa base de dados e vamos implementar este método com o código da sequência. Veja que pela assinatura, o método recebe 4 argumentos, que serão encaminhados para o método `createConnection()` de `SQLiteConnection (sqlite)` , são eles: nome da base/arquivo (`database`), se é ou não criptografada (`encrypted`), o modo para a encriptação da base (`mode`) e a versão da base de dados (`version`). Poderíamos otimizar o `encrypted` e `mode` no código com um `if` , mas aqui busco seguir a documentação oficial do plugin. Detalhe: será preciso um import para `SQLiteDBConnection` e o VS Code lhe ajuda nisso.

```
async createConnection(database: string, encrypted: boolean,
    mode: string, version: number
): Promise<SQLiteDBConnection> {
    if (this.sqliteConnection != null) {
        try {
            const db: SQLiteDBConnection = await
this.sqliteConnection.createConnection(
                database, encrypted, mode, version);
            if (db != null) {
                return Promise.resolve(db);
            } else {
                return Promise.reject(new Error(`Erro ao criar a
conexão`));
            }
        } catch (err) {
            return Promise.reject(new Error(err));
        }
    } else {
        return Promise.reject(new Error(`Nenhuma conexão disponível para
${database}`));
    }
}
```

Com o que temos, podemos começar com o consumo destes serviços por nossa aplicação além do acesso à plataforma de execução dela. Vamos então ao `app.component.ts`. Vamos começar injetando este nosso serviço e o acesso à plataforma ao construtor de `AppComponent`. Veja o código dele na sequência. Será preciso importar `DatabaseService` e `Platform`.

```
constructor(
    private storage: Storage,
    private platform: Platform,
    private databaseService: DatabaseService,
    ) {  }
```

Vamos agora à implementação do método de inicialização que indicaremos para nosso App. Veja-o na sequência, mas observe que temos uma declaração de variável, que deverá ser inserida antes do

construtor. Ela apenas refletirá, o sucesso ou não, no processo de inicialização do plugin. E, após essa declaração, temos o método `initializeApp`, que inicializará nosso plugin e criará a conexão com a base de dados que, caso o arquivo não exista, será criado. Essa variável tem o fim didático para um controle que você possa utilizar em sua aplicação.

```
// Antes do construtor
private initPlugin: boolean;

// Nosso método
async initializeApp() {
    this.platform.ready().then(async () => {
      this.databaseService.initializePlugin().then(async (ret) => {
        try {
          const db = await
this.databaseService.createConnection("oficina", false, "no-encryption",
1);
          this.initPlugin = ret;
        } catch (err) {
          console.log(`Error: ${err}`);
          this.initPlugin = false;
        }


         console.log('Status da inicialização do plugin: ' +
this.initPlugin);
      });
   });
}
```

Agora, dentro do construtor, precisamos invocar esse método implementado. Basta inserirmos a invocação a ele entre as chaves do construtor, como no código a seguir.

```
this.initializeApp();
```

Nós podemos, por curiosidade apenas, executarmos nossa aplicação, mas eu sei que você pode estar se perguntando: "E as tabelas?". Aí

te respondo: calma, estamos em um processo. Vamos primeiro à execução. Lembre-se que agora ela precisa ser em um dispositivo ou emulador, pois no navegador não funcionará da maneira que estamos fazendo, mas este plugin permite também o uso em navegadores. Mas deixo para você esta implementação. Vamos lá. No terminal, digite as instruções a seguir.

```
npm run build
ionic capacitor run android -l --external
```

Como estamos utilizando o dispositivo/emulador, vamos querer algum tipo de depuração em relação aos logs que registramos no código. Desta maneira, quando o App estiver em execução em seu dispositivo ou emulador, no Chrome, digite no campo de URL o endereço a seguir, que vimos no Capítulo 2.

```
chrome://inspect/#devices/
```

Na página que se abre, precisamos inspecionar a execução do App em nosso dispositivo, sendo assim, clique no link em destaque na figura a seguir. Aproveito e coloco, na mesma figura, a página aberta pelo link, com a execução de nosso App espelhado no navegador e com a console ativa, exibindo nossos logs. Aqui uma observação: infelizmente, com este componente, não consegui fazer que a cada reload a conexão fosse reaproveitada, criada ou reaberta, o que, infelizmente, me levou à necessidade de a cada alteração, parar a execução da aplicação e executar novamente. Essa é uma coisa bem chata. Outra maneira é sempre mudar o nome da base de dados antes de um live reloading.

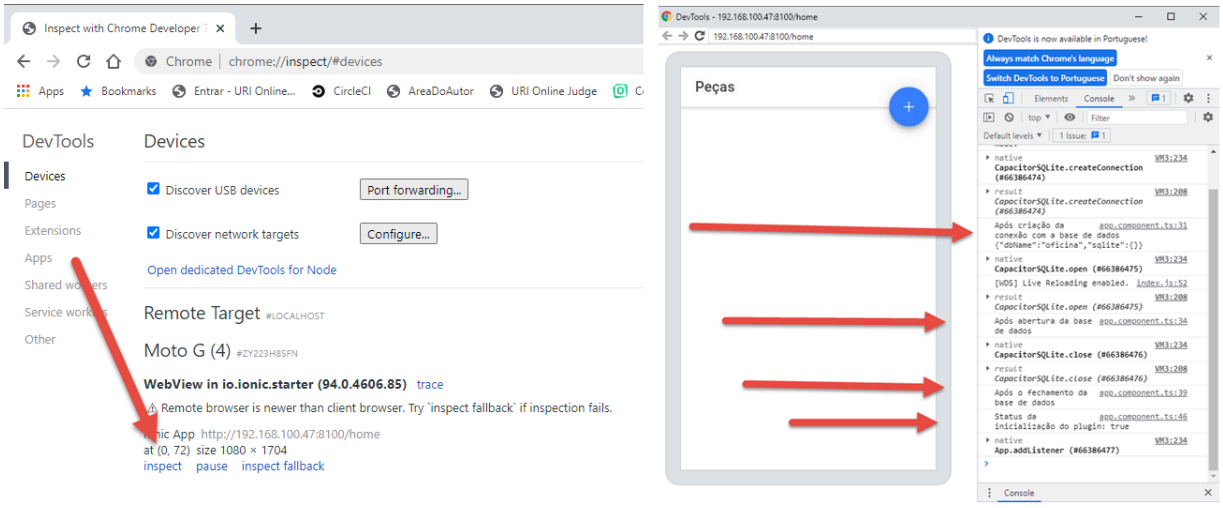


Figura 5.5: Depurando o App via Chrome.

Tudo certo e maravilhoso, mas ainda não temos nada em nossa base de dados, nenhuma tabela, muito menos os dados. Vamos então trabalhar a criação de nossa primeira tabela. Vamos, na pasta `services`, criar um novo arquivo, chamado `database.statements.ts` e nele vamos inserir o código a seguir. Observe que apenas declaramos uma string, chamada `createSchema`, com instruções de criação de uma tabela e de um índice para ela.

```
export const createSchema: string = `
CREATE TABLE IF NOT EXISTS ordensdeservico (
    ordemdeservicoid TEXT primary key NOT NULL,
    clienteid TEXT NOT NULL,
    veiculo TEXT NOT NULL,
    dataehoraentrada DATETIME NOT NULL,
    dataehoratermino DATETIME,
    dataehoraentrega DATETIME
);

CREATE INDEX IF NOT EXISTS ordensdeservico_index_ordemdeservicoid ON
ordensdeservico (ordemdeservicoid);
PRAGMA user_version = 1;
`;
```

Agora, em nosso `database.service.ts`, vamos criar um método, de acordo ao código a seguir. Fique atento(a) ao import para o código anterior e observe que o método recebe o `SQLiteDBConnection` na variável `db`, abre a conexão recebida, executa a instrução que criamos anteriormente, fecha a conexão com a base e retorna a rejeição ou resolução do método, de acordo aos resultados obtidos. Veja que o método é privado.

```
private async createSchema(db: SQLiteDBConnection): Promise<void> {
    await db.open();

    let createSchemma: any = await db.execute(createSchema);

    await db.close();

    if (createSchemma.changes.changes < 0) {
        return Promise.reject(new Error("Erro na criação das tabelas"));
    }

    return Promise.resolve();
}
```

Precisamos agora invocar este método e o faremos no método `createConnection()`, que já temos em nosso serviço e essa invocação será antes do `Promise.resolve(db)`, que temos na verificação de não nulidade da conexão. Para facilitar, trago a implementação com este trecho de código, na sequência.

```
if (db != null) {
    await this.createSchema(db);
    return Promise.resolve(db);
} else {
    return Promise.reject(new Error(`Erro ao criar a conexão`));
}
```

Vamos novamente executar nossa aplicação, observando o console no Chrome, verificando se nossa tabela é realmente criada. Você precisa interromper a execução da aplicação e executá-la novamente, para que uma nova conexão com a base de dados seja

realizada ok? Quando digo "criação de uma nova conexão", não significa que um novo arquivo de base de dados será criado, mas sim a conexão lógica com ele. É assim que o plugin trabalha.

Até aqui temos implementadas a criação da base de dados e de nossa primeira tabela no SQLite e, seguindo nosso último exemplo de testes, começaremos com a listagem de ordens de serviço, mas ainda não temos nenhuma. Podemos pensar em implementar a página de adição e alteração, como já temos para os demais modelos de negócio, mas vamos pensar em popular nossa base com alguns dados para testes. Veja então o método a seguir, que devemos implementar no arquivo `database.service.ts` que estávamos trabalhando.

```
private async populateDatabase(db: SQLiteDBConnection): Promise<void> {
    let returnQuery = await db.query("select COUNT(ordemdeservicoid) as
qtdeOS from ordensdeservico;");

    if (returnQuery.values[0].qtdeOS === 0) {
        let sqlcmd: string =
            "INSERT INTO ordensdeservico (ordemdeservicoid, clienteid,
veiculo, dataehoraentrada, dataehoratermino) VALUES (?,?,?,?,?)";
        let values: Array<any> = [Guid.create().toString(),
Guid.create().toString(), 'ABC-1235', Date.now(), null];

        let returnInsert = await db.run(sqlcmd, values);
        if (returnInsert.changes < 0) {
            return Promise.reject(new Error("Erro na inserção da ordem de
serviço"));
        }
    }

    return Promise.resolve();
}
```

Observe que realizamos uma consulta que nos retornará a quantidade de registros que temos na tabela que estamos criando nesta seção. Caso não existam registros, inserimos então os iniciais.

Optei por apenas um, mas você pode redundar o código para inserir mais. Novamente reforço que não entrarei em detalhes sobre SQL. Mas quando este método será invocado? A resposta é: logo após a criação de nosso schema, implementado anteriormente no método `createSchema()`. Será necessário importar o `Guid`. Para facilitar, veja essas instruções na seguir.

```
let createSchemma: any = await db.execute(createSchema);
await this.populateDatabase(db);
```

Pode haver um momento em que você precise apagar o arquivo da base de dados criado pelo plugin. Há um processo não trivial, de acesso ao emulador/dispositivo e então remoção do arquivo. É possível até abri-lo em um aplicativo como o `DBBrowser` para acompanhar os registros. Recomendo aqui, para isso, a leitura do link https://aboutreact.com/see-saved-data-of-the-sqlite-database-in-device/.

Um workaround para trabalhar o livereload do Ionic com este plugin, é a cada vez que o isso for necessário, mudar o nome da base enviado no `createConnection`. Isso é chato também, mas menos trabalhoso em relação em parar a aplicação e executar de novo.

O plugin fornece métodos para remoção do arquivo da base de dados, mas para o nosso contexto não vejo muito objetivo nele.

Precisamos configurar o `DatabaseService` para ser injetado nas classes que assim precisarem dele. Desta maneira, no `app.module.ts`, em `provider` de `@NgModule`, insira `DatabaseService`.

5.7 A interface com o usuário para as Ordens de Serviço

Para iniciarmos a etapa de criação de nossa interface com o usuário, estamos adotando a prática de criarmos inicialmente a página de

listagem dos dados de nosso modelo de negócio. Desta maneira, teremos uma repetição de implementação em relação à técnica aqui adotada, mas a ideia é trazer o código para você, para facilitar o acompanhamento. Assim, as próximas subseções terão estes códigos.

A classe de modelo

Como estamos trabalhando sempre com uma interface que representa nosso modelo de negócio, vamos criar, na pasta `src/app/models`, um arquivo chamado `ordemdeservico.model.ts`, tal qual o código a seguir. Observe que inicialmente não temos como propriedade todos os campos que criamos em nossa tabela no SQLite.

```
export interface OrdemDeServico {
    ordemdeservicoid: string;
    clienteid: string;
    veiculo: string;
    dataehoraentrada: Date;
}
```

O Service para ordens de serviço

Na pasta `src/app/services`, vamos criar um arquivo chamado `ordensdeservico.service.ts`, tal qual apresentamos na sequência. Optei em não fazer uso do try...catch, mas reforço a importância dele e recomendo que você sempre o utilize em situações previsíveis para erros.

```
import { Injectable } from '@angular/core';
import { OrdemDeServico } from '../models/ordemdeservico.model';
import { DatabaseService } from './database.service';

@Injectable({
    providedIn: 'root'
})
export class OrdensDeServicoService {
```

```
    constructor(
        private databaseService: DatabaseService
    ) { }

    public async getAll() {
        const db = await
this.databaseService.sqliteConnection.retrieveConnection(databaseName);

        db.open();

        let returnQuery = await db.query("SELECT * FROM
ordensdeservico;");

        let ordensdeservico: OrdemDeServico[] = [];
        if (returnQuery.values.length > 0) {
            for (let i = 0; i < returnQuery.values.length; i++) {
                const ordemdeservico = returnQuery.values[i];
                console.log(`OS> ${ordemdeservico}`);
                ordensdeservico.push(ordemdeservico);
            }
        }

        return ordensdeservico;
    }
}
```

Verifique que injetamos um `DatabaseService` para a classe atual, que também é um `Service`. Implementamos um método básico para recuperação de todos os dados da tabela `ordensdeservico`, com todos os campos. A população das propriedades será automática, uma vez que o nome dos campos é o mesmo delas. Isso será realizado para cada registro recuperado. Veja que antes de executarmos o `query()`, abrimos a base de dados com `db.open()`.

Observe que no código anterior, estamos utilizando novamente o nome para a base de dados, "oficina". Vamos adaptar isso para uma constante e mudar no `app.component.ts` e no método criado há pouco, pois quando precisarmos alterar, alteramos em um só local.

Para isso, no arquivo `database.statements.ts`, insira o código a seguir, logo no início.

```
export const databaseName: string = 'oficina';
```

Agora, nos lugares onde usamos a string, substituímos pelo nome da constante, realizando as devidas importações. Para ajudar, veja na sequência o código dessa instrução, que implementamos há pouco no método `getAll()` e, na sequência, no `app.component.ts`.

```
const db = await
this.databaseService.sqliteConnection.retrieveConnection(databaseName);
```

Vamos agora implementar nossa página de listagem de ordens de serviço. Veja a seguir a instrução a ser executada no terminal e na sequência as orientações para ajustes nos arquivos.

```
ionic generate page pages/ordensdeservico/ordensdeservico-listagem
```

Classe controladora para a página de listagem

Na mesma pasta criada com o `generate`, no arquivo `ordensdeservico-listagem.page.ts`, vamos implementar o código a seguir. Temos a propriedade que será populada com os objetos recuperados, a injeção do Service e o método `ionViewWillEnter()`.

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { IonList } from '@ionic/angular';
import { OrdemDeServico } from 'src/app/models/ordemdeservico.model';
import { OrdensDeServicoService } from
'src/app/services/ordensdeservico.service';
import { ToastService } from 'src/app/services/toast.service';

@Component({
  selector: 'app-ordensdeservico-listagem',
  templateUrl: './ordensdeservico-listagem.page.html',
  styleUrls: ['./ordensdeservico-listagem.page.scss'],
})
export class OrdensdeservicoListagemPage implements OnInit {
```

```
  public ordensDeServico: void | OrdemDeServico[] = [];
  @ViewChild('slidingList') slidingList: IonList;

  constructor(
    private ordensdeservicoService: OrdensDeServicoService,
    private toastService: ToastService
  ) { }

  ngOnInit() {
  }

  async ionViewWillEnter() {
    const oss = await this.ordensdeservicoService.getAll();
    this.ordensDeServico = oss;
  }
}
```

Template para a página de listagem

No arquivo `ordensdeservico-listagem.page.html` criado, vamos inserir nele o código a seguir. Veja aqui que, com exceção dos dados do modelo de negócio, toda a implementação é semelhante ao que fizemos para `Peças` . Ainda não estamos incluindo comportamentos relacionados a seleção e inserção. Faremos isso na sequência.

```
<ion-header>
    <ion-toolbar>
      <ion-title>Ordens de serviço</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content padding-left padding-top>
  <ion-list #slidingList>
    <ion-item-sliding *ngFor="let ordemDeServico of ordensDeServico">
      <ion-item>
        <ion-label>
          <h2>{{ordemDeServico.veiculo}}</h2>
          <p style="text-align:right">Entrada:
{{ordemDeServico.dataehoraentrada | date: 'dd/MM/yyyy'}}</p>
```

```
        </ion-label>
      </ion-item>

      <ion-item-options side="end">
        <ion-item-option color="danger">
          <ion-button color="danger" no-padding>
              <ion-icon name="trash"></ion-icon>
              Remover
          </ion-button>
        </ion-item-option>
      </ion-item-options>
    </ion-item-sliding>
  </ion-list>


  <ion-fab vertical="top" horizontal="end" slot="fixed" edge="true">
    <ion-fab-button>
      <ion-icon name="add"></ion-icon>
    </ion-fab-button>
  </ion-fab>
</ion-content>
```

Observe a instrução que exibirá a `dataehoraentrada`. Estamos utilizando um `Pipe` do Angular, que é herdado pelo Ionic. Ele permite a transformação de um dado para uma formatação específica, em nosso caso um `date` no formato `dd/MM/yyyy`.

Os demais arquivos criados, neste momento, não carecem de implementações específicas, sendo o código gerado para eles, o necessário para este momento.

5.8 A aplicação em execução

Se executarmos nossa aplicação agora, a página a ser exibida ainda será a de peças e queremos a de ordem de serviço. Precisamos mudar o `app-routing.module.ts`, da mesma maneira que estamos

fazendo sempre, que é ajustando o `redirectTo` de `path:''` para `ordensdeservico-listagem`.

Já sabemos que não podemos testar esta funcionalidade nos pseudoemuladores. Desta maneira, você pode escolher conectar seu dispositivo em seu computador, ou executar um emulador. Com tudo isso ajustado, podemos novamente executar nossa aplicação e a instrução para isso está na sequência.

```
ionic capacitor run android -l --external
```

Com isso, será executado todo o processo necessário para compilação referente à criação da aplicação para o Android, inicialização do emulador, distribuição dela e execução. Você verá sua aplicação como mostra a figura a seguir. Aproveito para mostrá-la no emulador e na tela de console do Chrome, como também vimos no capítulo 2, o que nos possibilita depurar a aplicação.

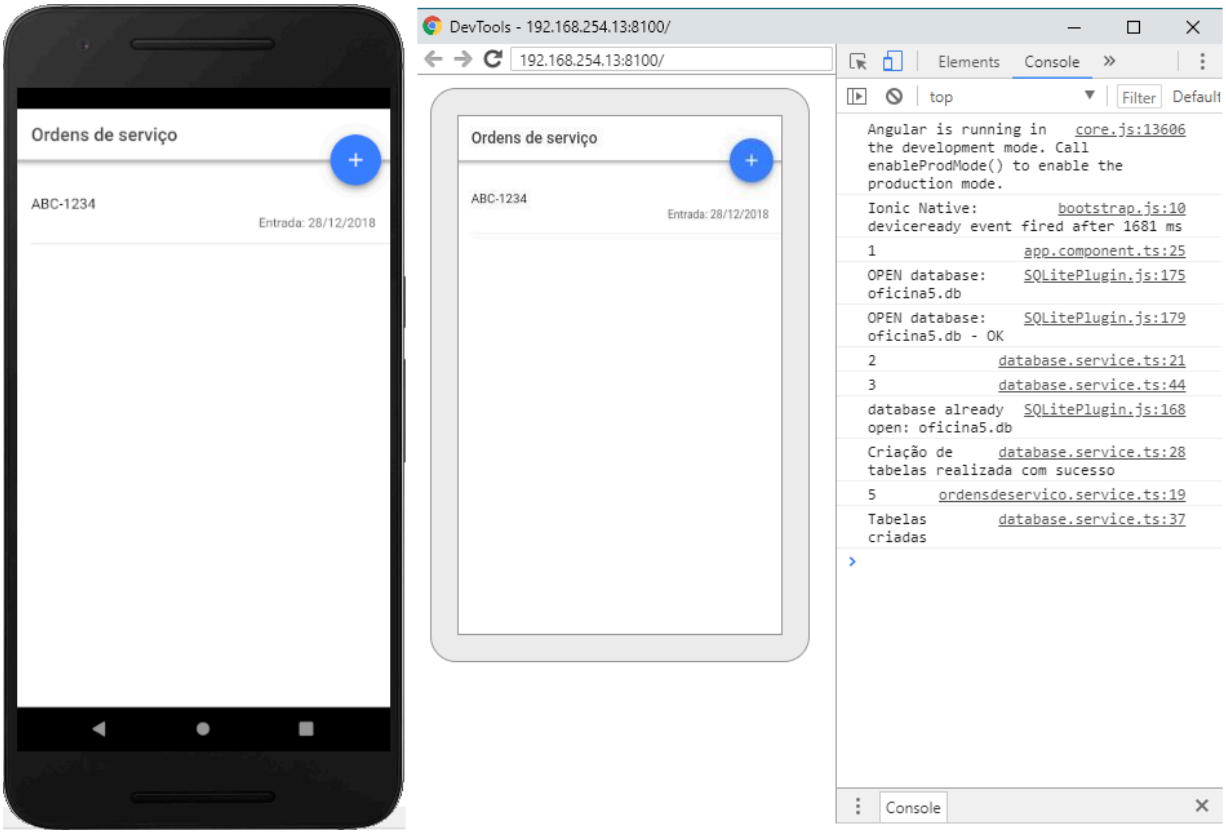


Figura 5.6: Aplicação Android em execução no emulador e console do Chrome

Vamos agora ao iOS? Vamos para a mesma estratégia que utilizamos há pouco para o Android, sem a necessidade de abrirmos o XCode. Se for utilizar um dispositivo no iOS, pode ser que seja necessário realizar toda a configuração de equipe no XCode na primeira execução, como visto no capítulo 2.

```
ionic capacitor run ios -l --external
```

Com o sucesso na execução das instruções anteriores, podemos ver nossa aplicação funcionando, como na figura a seguir.

12:25
Ordens de serviço
ABC-1234
Entrada: 29/12/2018

Figura 5.7: Aplicação iOS em execução no emulador do iPhone X

Ainda, com o que temos, se você realizar uma alteração no código, que force a recarga, como já dito, haverá erros no momento da criação de conexão com a base de dados. Desta maneira, precisará sempre parar sua aplicação e executá-la de novo, ou mudar o nome da base de dados aribuída a constante que criamos.

Conclusão

Chegamos ao final deste interessante capítulo. Colocamos em prática tudo que vimos nos anteriores e adicionamos o recurso de persistência local, tanto por meio do Ionic Storage, que é nativo na plataforma, como pelo banco de dados SQLite, por meio de um plugin para o Capacitor.

Criamos serviços para criação e obtenção do banco de dados e suas tabelas, assim como a inserção de dados que possam popular nossa base para testes. Reforçamos também o uso de emuladores para a execução das aplicações, agora diretamente pelo Ionic CLI, fazendo uso do Capacitor.

Nossos testes com o SQLite se limitaram a uma listagem de registros existentes em uma tabela, que em nosso caso foi a de Ordens de Serviço. Não concluímos o CRUD neste capítulo, mas o faremos no próximo capítulo, onde abordaremos recursos que serão necessários para as funcionalidades faltantes.

Capítulo 6
Sidemenu, CRUD com SQLite e o componente Select

Finalizamos o capítulo anterior com a criação e acesso a uma base de dados SQLite para manutenção e consulta em dados relacionados ao registro do atendimento de um veículo ao entrar em nossa oficina. No início do capítulo 5, vimos um CRUD completo com o Ionic Storage, mas não completamos o CRUD com o SQLite, pois isso deixaria o capítulo muito extenso.

Agora, neste nosso sexto capítulo, daremos sequência e conclusão ao CRUD de Ordens de Serviço, conheceremos um estilo de menu para acesso às páginas da aplicação e veremos também um componente que tenha a funcionalidade de um ComboBox.

Se lembrarmos bem, nossa tabela de Ordem de Serviço possui uma associação com clientes, por meio de um campo chamado `clienteid`, que representa a chave estrangeira para a tabela de clientes, mas nós também não criamos esta tabela no capítulo anterior. Trabalharemos tudo isso neste capítulo.

6.1 Criação do Sidemenu

É interessante que uma aplicação com diversas funcionalidades, representadas por páginas, possua um ponto de início, um menu, que permita o acesso a essas funcionalidades. Existem algumas variações para isso. Os menus em aplicações móveis têm adotado a característica conhecida como Sidemenu, que é uma página que não é exibida em sua totalidade e que pode ser deslizada para a esquerda (normalmente) para ser ocultada, ou então, puxada para a direita para ser exibida.

Vamos começar nossa implementação criando uma página que será a exibida por padrão quando a aplicação iniciar. Normalmente esta página é conhecida como `Dashboard` e vamos utilizar este nome em sua criação. Lá no terminal, tal qual fizemos as últimas gerações de páginas, execute a instrução a seguir. Vamos continuar o projeto do capítulo anterior, ok?

```
ionic generate page pages/dashboard
```

Nós vamos trabalhar para que uma imagem seja exibida como background do Dashboard, apenas para termos essa opção de como aplicar esta funcionalidade e teremos também um `Hamburguer Menu`, onde as opções de acesso ao usuário serão exibidas. Veja a figura a seguir, que traz estas duas interfaces que teremos após nossa implementação.

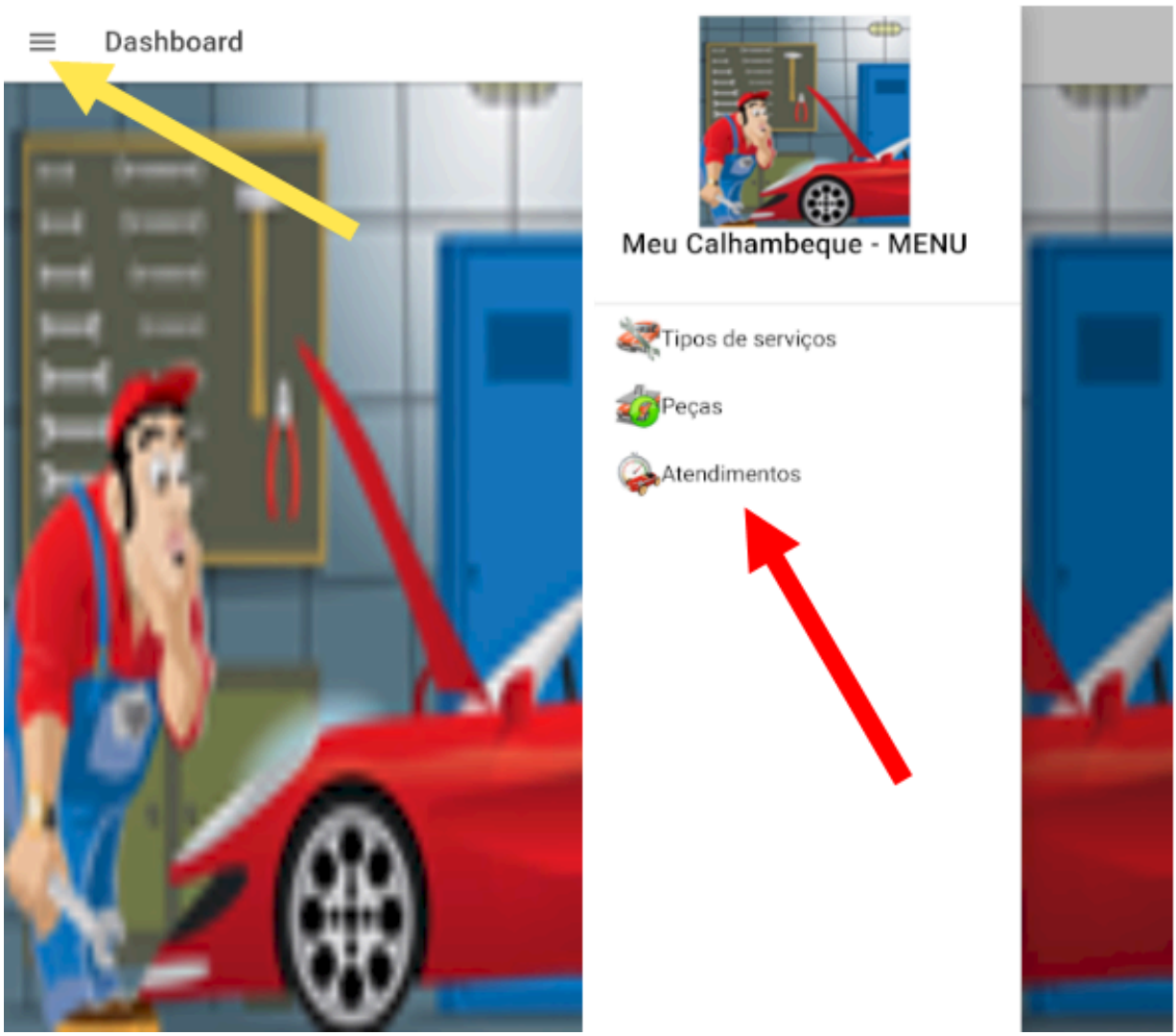


Figura 6.1: Dashboard com Hamburguer Menu e Opções de acesso

Vamos agora criar um estilo para que a imagem que utilizarmos como background possa ser exibida. Esta é certamente uma das muitas possibilidades de implementar esta funcionalidade. Sendo assim, no arquivo `dashboard.page.scss` implemente o seguinte código:

```
.background {
  --background: url("/assets/imgs/logo.png") 0 0/100% 100% no-repeat
 }
```

Agora, no arquivo `dashboard.page.html` , o código que é apresentado na sequência. O `<ion-buttons>` refere-se ao Hambuguer Menu, que implementaremos daqui a pouco, mas já prevemos aqui sua

exibição. Note também no `<ion-content>` que definimos `fullscreen` com `true` e utilizamos a classe de estilos que implementamos no código anterior.

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
      <ion-menu-button></ion-menu-button>
    </ion-buttons>
    <ion-title>Dashboard</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content fullscreen="true" class="background">
</ion-content>
```

Os arquivos SCSS e HTML para o Sidemenu

Vamos agora começar nossa implementação para a criação de nosso Sidemenu, que trará o acesso que precisamos para as funcionalidades de nossa aplicação.

Trabalharemos inicialmente no nível do aplicativo e começaremos com o arquivo `app.component.scss`, onde implementaremos o código a seguir.

```
.active-item {
    border-left: 8px solid var(--ion-color-primary);
}
```

Na sequência, no arquivo `app.component.html`, vamos implementar o código responsável pela geração do Sidemenu. Veja-o na listagem que segue e veja que já utilizamos a classe anterior no `<ion-item>`.

```
<ion-app>
  <ion-split-pane contentId="main-content">
    <ion-menu contentId="main-content" type="overlay">
      <ion-content>
        <ion-list id="inbox-list">
```

```
          <ion-img style="width: 300px; height: 150px"
src="assets/imgs/logo.png" padding>
          </ion-img>
          <ion-title>Meu Calhambeque - MENU</ion-title>
          <ion-item-divider>
          </ion-item-divider>
          <ion-menu-toggle auto-hide="false" *ngFor="let p of pages; let i
= index">
            <ion-item (click)="selectedIndex = i" routerDirection="root"
[routerLink]="[p.url]" lines="none"
              detail="false" [class.active-item]="selectedIndex == i">
              <ion-img [src]="p.icon" style="width: 32px; height: 32px;">
</ion-img>
              <!-- <ion-icon slot="start" [ios]="p.icon'" [md]="p.icon-
sharp'"></ion-icon> -->
              <ion-label>{{ p.title }}</ion-label>
            </ion-item>
          </ion-menu-toggle>
        </ion-list>
      </ion-content>
    </ion-menu>
    <ion-router-outlet id="main-content"></ion-router-outlet>
  </ion-split-pane>
</ion-app>
```

O primeiro componente que temos em nosso template, após o `<ion-app>` é o `<ion-split-pane>`. A documentação do Ionic traz a definição dele como um componente que torna possível a criação de um layout conhecido por `multi-view`, ou seja, a visualização de mais de uma página na mesma tela, muito comum em aplicações originárias no iPad, possibilitando o uso de menus.

Com o uso do `SplitPane`, se o tamanho da tela do dispositivo é menor que um determinado valor, o componente ocultará o menu, dando preferência à visão atualmente ativa. De acordo com a documentação, este componente é útil quando se desenvolve aplicações que serão executadas em dispositivos móveis, como tablets e telefones, ou para executar no navegador.

Dentro do `<ion-split-pane>` começamos a definir nosso menu, fazendo uso do componente `<ion-menu>`, que é o responsável pela criação dos componentes que representarão as opções ao usuário, assim como permitir que a página de menu seja deslizada, tanto para ocultá-lo, como para exibi-lo.

Observe que no primeiro `<ion-img>`, dentro do `<ion-list>`, que conterá nossas opções, estamos definindo uma imagem que será utilizada como logomarca para a aplicação. Lembre-se de ter esta imagem fisicamente em seu projeto e nós a utilizamos também como background para a página de Dashboard.

Dentro do componente `<ion-menu>`, temos o atributo `contentId`, recebendo o valor `main-content`, que será utilizado pelo componente responsável pela renderização do conteúdo das páginas acessadas pelas opções disponibilizadas pelo menu. Antecipando, este componente é o `<ion-router-outlet>` que está ao final do código. Já comentaremos sobre ele.

No componente `<ion-list>`, temos um novo, o `<ion-menu-toggle>`, que será responsável por gerenciar a abertura das páginas de conteúdo que a aplicação acessará a partir do menu. Usamos este mesmo valor no `<ion-split-pane>`. Ele garante a alternância entre visível ou não para o menu. Temos no controle o `*ngFor`, que percorrerá uma coleção que definiremos na classe do template, que possibilitará a renderização dos itens do menu, por meio do conteúdo que temos no `<ion-item>`, onde temos um `<ion-img>` e um `<ion-label>` para cada item. Observe os bindings para estes componentes.

Ao final, temos o `<ion-router-outlet>`. Já o conhecemos lá do capítulo 2, onde vimos e estudamos a estrutura de uma aplicação Ionic. Observe que definimos o nome `main-content` para ele, o id `main-content` que é o que utilizamos no `<ion-menu>`.

A classe TypeScript para nosso template

Com nosso SCSS e HTML implementados, podemos verificar a implementação para a classe que responderá pela renderização dele. Vamos então no arquivo para ela, implementar o código a seguir. Isso será no `app.component.ts`, que já temos implementação. Desta maneira, verifique as orientações no código.

```
// No início da classe

pages = [
    {
      title: 'Tipos de serviços',
      url: '/tiposdeservicos',
      icon: '/assets/imgs/icon_tiposservicos.png'
    },
    {
      title: 'Peças',
      url: '/pecas',
      icon: '/assets/imgs/tab_pecas.png'
    },
    {
      title: 'Atendimentos',
      url: '/ordensdeservico-listagem',
      icon: '/assets/imgs/icon_atendimentos.png'
    }
];
```

Logo no início do código, temos a declaração de uma variável: `pages`, que é um array, com objetos que representam dados sobre as páginas, que também estamos utilizando no template, para a renderização das opções. Veja as imagens que precisaremos ter fisicamente em nossa aplicação. Observe o `icon`, o conteúdo entre parênteses, `main-content`, que é o nome que demos ao controle `<ion-router-outlet>`.

Para finalizarmos e vermos o menu em ação, no arquivo `ordensdeservico-listagem.page.html`, vamos inserir o código a seguir antes do `<ion-title>`, que está no início do código.

```
<ion-buttons slot="start">
  <ion-menu-button></ion-menu-button>
</ion-buttons>
```

6.2 A tabela de clientes na base de dados

No capítulo anterior, criamos um serviço específico para o acesso à base de dados, que foi implementado no arquivo `database.service.ts`. Neste serviço, temos o método `createSchema()`, onde invocamos uma constante com instruções para criamos a tabela `ordensdeservico`. Precisamos agora criar também a tabela `clientes`. Vamos renomear a constante `createSchema` para `createOrdensDeServicoTable`. Depois, no mesmo arquivo, vamos criar outra constante, tal qual o código a seguir.

```
export const createClientesTable: string = `
CREATE TABLE IF NOT EXISTS clientes (
    clienteid TEXT primary key NOT NULL,
    nome TEXT NOT NULL,
    email TEXT NOT NULL,
    telefone TEXT NOT NULL,
    renda REAL NOT NULL
);

CREATE INDEX IF NOT EXISTS clientes_index_clienteid ON clientes
(clienteid);
PRAGMA user_version = 1;
`;
```

Precisamos, agora que temos a instrução responsável pela criação desta nova tabela, executá-la e faremos isso no método `createSchema()`, veja a nova implementação para este método na sequência. Observe as alterações em relação às variáveis que recebem o resultado de `execute`, pois agora criaremos o schema para duas tabelas. Observe que antes criamos a tabela para clientes.

```
private async createSchema(db: SQLiteDBConnection): Promise<void> {
    await db.open();

    let createClienteSchemma: any = await db.execute(createClientesTable);
    let createOSSchemma: any = await
db.execute(createOrdensDeServicoTable);

    await this.populateDatabase(db);

    await db.close();
    console.log('Fechou conexão');
    return Promise.resolve();
}
```

Com a tabela de clientes criada, precisamos populá-la, e faremos isso no método `populateDatabase()` de nosso serviço. Antes de adaptarmos o método, eu extraí a instrução a seguir, que está no `SQ` do `INSERT` em `ordensdeservico` para logo no início do método. Esta decisão foi tomada para que o registro inserido em `ordensdeservico` possua o mesmo valor para a coluna `clienteid` que foi inserida na tabela `clientes` como chave primária. Isso nos garantirá o relacionamento. Substitua o código pelo `clienteID` .

```
const clienteID = Guid.create().toString();
```

Com base nas responsabilidades de cada método, verifique o código a seguir, que traz as instruções exclusivas para a criação da tabela `ordensdeservico` , que extraí do `populateDatabase()` . Verifique que em síntese, temos todo o código de `populateDatabase()` neste novo método, o `populateOrdensDeServico()` , só que ele recebe uma string para o `clienteID` e faz uso dessa variável no `INSERT` .

```
private async populateOrdensDeServico(db: SQLiteDBConnection,
clienteID:String): Promise<void> {
    let returnQuery = await db.query("select COUNT(ordemdeservicoid) as
qtdeOS from ordensdeservico;");

    if (returnQuery.values[0].qtdeOS === 0) {
        let sqlcmd: string =
```

```
            "INSERT INTO ordensdeservico (ordemdeservicoid, clienteid,
veiculo, dataehoraentrada, dataehoratermino) VALUES (?,?,?,?,?)";
        let values: Array<any> = [Guid.create().toString(), clienteID,
'ABC-1235', Date.now(), null];

        let returnInsert = await db.run(sqlcmd, values);
        if (returnInsert.changes < 0) {
            return Promise.reject(new Error("Erro na inserção da ordem de
serviço"));
        }
    }

    return Promise.resolve();
}
```

Vamos agora criar um novo método, com a funcionalidade semelhante ao anterior, mas agora tendo a tabela de clientes como responsabilidade. Verifique na sequência.

```
private async populateClientes(db: SQLiteDBConnection, clienteID: String):
Promise<void> {
    let returnQuery = await db.query("select COUNT(clienteid) as
qtdeClientes from clientes;");

    if (returnQuery.values[0].qtdeClientes === 0) {
        let sqlcmd: string =
            "INSERT INTO clientes (clienteid, nome, email, telefone,
renda) VALUES (?,?,?,?,?)";
        let values: Array<any> = [clienteID, 'Ambrózio',
'ambrozio@casadocodigo.com.br', '91234-5668', 123];

        let returnInsert = await db.run(sqlcmd, values);
        if (returnInsert.changes < 0) {
            return Promise.reject(new Error("Erro na inserção da
clientes"));
        }
    }
```

```
    return Promise.resolve();
}
```

Precisamos agora alterar o `populateDatabase()` para invocar estes dois novos métodos. Veja na sequência a nova implementação para este método.

```
private async populateDatabase(db: SQLiteDBConnection): Promise<void> {
    const clienteID = Guid.create().toString();

    await this.populateClientes(db, clienteID);
    await this.populateOrdensDeServico(db, clienteID);

    return Promise.resolve();
}
```

Você deve ter verificado que, na criação da tabela `ordensdeservico` que fizemos no capítulo anterior, eu não me preocupei em efetivar o relacionamento entre as tabelas, mas é algo que deve ser pensado e, para resolver este problema, vamos reescrever o `CREATE TABLE` para `ordensdeservico`, tal qual está na sequência. Observe a instrução `FOREIGN KEY`.

```
CREATE TABLE IF NOT EXISTS ordensdeservico (
    ordemdeservicoid TEXT primary key NOT NULL,
    clienteid TEXT NOT NULL,
    veiculo TEXT NOT NULL,
    dataehoraentrada DATETIME NOT NULL,
    dataehoratermino DATETIME,
    dataehoraentrega DATETIME,
    FOREIGN KEY (clienteid) REFERENCES clientes (clienteid) ON DELETE
CASCADE ON UPDATE NO ACTION
);
```

6.3 Service para clientes

Com a tabela de `clientes` criada na seção anterior, podemos antecipar a visão da necessidade de manipular estes dados, que serão recuperados, inseridos, alterados ou removidos da tabela. Isso nos remete à necessidade de termos nossa classe de modelo de negócio para clientes, e a implementaremos em nossa pasta `/src/app/models`, com o arquivo chamando `cliente.model.ts`, de acordo com o código:

```
export interface Cliente {
    clienteid: string;
    nome: string;
    email: string;
    telefone: string;
    renda: number;
    nascimento: Date;
}
```

Precisamos agora implementar a classe de serviços para `clientes` e um método inicial que nos retorne todos os clientes registrados na tabela, para que, em nosso registro de atendimento, possamos selecionar o cliente do veículo que entá entrando na oficina.

Na pasta `services` crie um arquivo chamado `clientes.service.ts` e insira nele o código apresentado na sequência. Observe que o código é semelhante ao que implementamos no capítulo anterior para `ordensdeservico`. Temos apenas uma diferença, que é a cláusula `ORDER BY`, no `SELECT`.

```
import { Injectable } from '@angular/core';
import { DatabaseService } from './database.service';
import { Cliente } from '../models/cliente.model';
import { databaseName } from './database.statements';

@Injectable({
    providedIn: 'root'
})
export class ClientesService {
    constructor(
        private databaseService: DatabaseService
```

```
    ) { }

    public async getAll() {
        const db = await
this.databaseService.sqliteConnection.retrieveConnection(databaseName)

        db.open();
        let returnQuery = await db.query("SELECT * FROM clientes ORDER BY
nome");
        db.close();


        if (returnQuery.values.length > 0) {
            let clientes: Cliente[] = [];
            for (let i = 0; i < returnQuery.values.length; i++) {
                const cliente = returnQuery.values[i];
                // console.log(`OS> ${ordemdeservico}`);
                clientes.push(cliente);
            }
            return clientes;
        }

        return [];
    }
}
```

6.4 A inserção de ordens de serviço

Vamos fazer uso uma vez mais da geração de código para a criação de nossos artefatos que serão responsáveis pela inserção e alteração

de uma ordem de serviço. No terminal, na pasta de nosso projeto, execute a instrução a seguir.

```
ionic generate page pages/ordensdeservico/ordensdeservico-add-edit
```

Como teremos registro de data e hora no atendimento e, como vimos no capítulo 3, o trabalho com data nos pede um pouco mais de esforço. Vamos assumir que o app agora só funciona em dispositivo e não mais em navegador, precisamos então instalar o plugin `DatePicker`, tal qual fizemos antes. Veja o código a seguir, que deve ser executado no terminal.

```
// Instalação do plugin
npm install cordova-plugin-datepicker

// Instalação do acesso nativo ao plugin
npm install @ionic-native/date-picker

// Instalação da dependência para uso do plugin
npm install @ionic-native/core
```

Precisamos adicionar `DatePicker` ao `providers` de `@NgModule` do `app.module.ts` e, com os artefatos criados, vamos adaptá-los de acordo com as nossas necessidades e faremos isso nas seções que se seguem.

O arquivo ordensdeservico-add-edit.page.html

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
      <ion-back-button defaultHref="ordensdeservico-listagem"></ion-back-
button>
    </ion-buttons>
    <ion-title>Ordem de Serviço</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content padding>
```

```
  <ion-item-divider color="light">
    <ion-grid no-padding>
      <ion-row no-padding>
        <ion-col col-3 align-self-center>
          <ion-img style="width: 50px; height: 50px;"
src="assets/imgs/icon_atendimentos.png"></ion-img>
        </ion-col>
        <ion-col col-9>
          <ion-label style="font-size: 30px">Dados do atendimento</ion-
label>
        </ion-col>
      </ion-row>
    </ion-grid>
  </ion-item-divider>

  <ng-container *ngIf="osForm">
    <form [formGroup]="osForm">
      <ion-item lines="none" [disabled]="!modoDeEdicao">
        <ion-label position="floating" class="label-input-
title">Cliente</ion-label>

        <ion-select placeholder="Selecione um cliente" name="clienteid"
formControlName="clienteid"
          style="margin-left: 3%;" [value]="ordemDeServico.clienteid">
          <ion-select-option *ngFor="let cliente of clientes"
[value]="cliente.clienteid">
            {{cliente.nome}}
          </ion-select-option>
        </ion-select>

      </ion-item>

      <ion-item lines="none" [disabled]="!modoDeEdicao">
        <ion-label position="floating" class="label-input-
title">Veículo</ion-label>
        <ion-input formControlName="veiculo" type="text" style="margin-
left: 3%;"></ion-input>
      </ion-item>
```

```
      <ion-item lines="none" class="ion-no-padding">
        <div (click)="selecionarDataEntrada()">
          <ion-grid class="ion-no-padding">
            <ion-row>
              <ion-col size="9">
                <ion-item lines="none">
                  <ion-label position="floating">Data entrada</ion-label>
                  <ion-input formControlName="dataentrada"></ion-input>
                </ion-item>
              </ion-col>
              <ion-col></ion-col>
              <ion-col>
                <ion-item lines="none">
                  <ion-button fill="clear" class="ion-no-margin;ion-no-
padding">
                    <ion-icon icon="calendar"></ion-icon>
                  </ion-button>
                </ion-item>
              </ion-col>
            </ion-row>
          </ion-grid>
        </div>
      </ion-item>

      <ion-item lines="none" class="ion-no-padding">
        <div (click)="selecionarHoraEntrada()">
          <ion-grid class="ion-no-padding">
            <ion-row>
              <ion-col size="9">
                <ion-item lines="none">
                  <ion-label position="stacked">Hora entrada</ion-label>
                  <ion-input formControlName="horaentrada"></ion-input>
                </ion-item>
              </ion-col>
              <ion-col></ion-col>
              <ion-col>
                <ion-item lines="none">
                  <ion-button fill="clear" class="ion-no-margin;ion-no-
padding">
```

```
                    <ion-icon icon="calendar"></ion-icon>
                  </ion-button>
                </ion-item>
              </ion-col>
            </ion-row>
          </ion-grid>
        </div>
      </ion-item>

      <ion-button *ngIf="!modoDeEdicao" shape="round" color="primary"
expand="block" padding>
        Alterar dados
      </ion-button>

      <div *ngIf="modoDeEdicao" no-padding>
        <ion-grid>
          <ion-row>
            <ion-col col-6>
              <ion-button shape="round" color="success" size="small"
padding expand="block">
                Gravar
              </ion-button>
            </ion-col>

            <ion-col col-6>
              <ion-button shape="round" color="warning" size="small"
padding expand="block">
                Cancelar
              </ion-button>
            </ion-col>
          </ion-row>
        </ion-grid>
      </div>
    </form>
  </ng-container>
</ion-content>
```

Uma primeira mudança no código anterior refere-se ao componente que representa o botão para retorno à página anterior. Veja-o em

detalhe na sequência. Observe a URL alvo. É nossa listagem de ordens de serviço já registradas.

```
<ion-back-button defaultHref="ordensdeservico-listagem"></ion-back-button>
```

Observe a variável `modoDeEdicao`, que trabalharemos no TypeScript, para definir e identificar o estado da aplicação em relação ao atendimento em tela.

A próxima novidade é o componente `<ion-select>`, que será o responsável por exibir os clientes já registrados, para que possamos selecionar o proprietário do veículo que está entrando na oficina. Esta pode não ser a melhor opção, pois o menos custoso para a aplicação seria termos uma janela de pesquisa, onde o usuário possa digitar uma parte do nome e então uma seleção com este valor seja realizada. Faremos isso mais à frente, mas a ideia aqui é a de apresentar este novo controle.

```
<ion-select placeholder="Selecione um cliente" name="clienteid"
formControlName="clienteid" style="margin-left: 3%;"
[value]="ordemDeServico.clienteid">
  <ion-select-option *ngFor="let cliente of clientes"
[value]="cliente.clienteid">
    {{cliente.nome}}
  </ion-select-option>
</ion-select>
```

Temos um `*ngFor` percorrendo uma coleção de clientes, que logo conheceremos. A propriedade `value`, que está ligada à propriedade `clienteid` de cada objeto obtido da coleção, permitirá obtermos o valor do ID do cliente selecionado, assim como temos o nome do cliente, que será utilizado para exibição na janela que exibirá os clientes registrados. Já o `[value]` no `<ion-select>` refere-se ao cliente da ordem de serviço, para o caso de alteração de um dado, que logo veremos também, mas trago antecipado aqui.

Depois, para a data e hora de entrada, estamos reutilizando a mesma estrutura utilizada no capítulo 3. Temos um bloco específico

para a data de entrada e um para hora de entrada.

A classe TypeScript para nosso template

Como você poderá verificar, não temos nada de novo, são funcionalidades que já implementamos em capítulos anteriores. Estou utilizando o nome `OrdensDeServicoAddEditPage` para a classe, mas, na geração do código, no `ordensdeservico-add-edit.module.ts`, o `De` ficou `de`. Precisei ajustar no arquivo. Tive que fazer o mesmo no `ordensdeservico-add-edit-routing.module.ts`. Verifique no código os métodos específicos para a data e hora de entrada. Há um pouco de redundância e que poderia ser melhorado com uma refatoração, mas te deixo este desafio.

```
import { Component, OnInit } from '@angular/core';
import { OrdemDeServico } from 'src/app/models/ordemdeservico.model';
import { FormGroup, FormBuilder, Validators } from '@angular/forms';
import { Guid } from 'guid-typescript';
import { Cliente } from 'src/app/models/cliente.model';
import { ClientesService } from 'src/app/services/clientes.service';
import { DatePicker } from '@ionic-native/date-picker/ngx';
import { Platform } from '@ionic/angular';


@Component({
  templateUrl: './ordensdeservico-add-edit.page.html'
})
export class OrdensDeServicoAddEditPage implements OnInit {
  private ordemDeServico: OrdemDeServico;
  public modoDeEdicao = false;
  public osForm: FormGroup;
  public clientes: void | Cliente[] = [];

  constructor(
    private formBuilder: FormBuilder,
    private clientesService: ClientesService,
    private datePicker: DatePicker,
    private platform: Platform,
  ) { }
```

```
  async ngOnInit() {
  }

  async ionViewWillEnter() {
    const clientes = await this.clientesService.getAll();
    this.clientes = clientes;

    this.ordemDeServico = {
      ordemdeservicoid: Guid.createEmpty().toString(),
      clienteid: Guid.createEmpty().toString(),
      veiculo: '',
      dataehoraentrada: new Date()
    };
    this.modoDeEdicao = true;

    this.osForm = this.formBuilder.group({
      ordemdeservicoid: [this.ordemDeServico.ordemdeservicoid],
      clienteid: [this.ordemDeServico.clienteid, Validators.required],
      veiculo: [this.ordemDeServico.veiculo, Validators.required],
      dataentrada: [{ value:
this.ordemDeServico.dataehoraentrada.toLocaleDateString(), disabled:
!this.modoDeEdicao }, Validators.required],
      horaentrada: [{ value:
this.ordemDeServico.dataehoraentrada.toLocaleTimeString(), disabled:
!this.modoDeEdicao }, Validators.required],
      dataehoraentrada: [this.ordemDeServico.dataehoraentrada]
    });
  }

  selecionarDataEntrada() {
    if (!this.modoDeEdicao) {
      return;
    }

    this.platform.ready().then(() => {
      if (this.platform.is('capacitor')) {
        this.datePicker.show({
          date: new Date(),
```

```
          mode: 'date',
          locale: 'pt-br',
          doneButtonLabel: 'Confirmar',
          cancelButtonLabel: 'Cancelar'
        })
          .then(
            data => {

this.osForm.controls.dataentrada.setValue(data.toLocaleDateString());
              this.osForm.controls.horaentrada.setValue('');
            }
          );
      } else {
        // instruções para execução no navegador
      }
    });
  }

  selecionarHoraEntrada() {
    if (!this.modoDeEdicao) {
      return;
    }

    this.platform.ready().then(() => {
      if (this.platform.is('capacitor')) {
        this.datePicker.show({
          date: new Date(),
          mode: 'time',
          locale: 'pt-br',
          doneButtonLabel: 'Confirmar',
          cancelButtonLabel: 'Cancelar'
        })
          .then(
            data => {
              const dataEHoraEntrada = new
Date(this.osForm.controls.dataehoraentrada.value)

              dataEHoraEntrada.setHours(data.getHours());
              dataEHoraEntrada.setMinutes(data.getMinutes());
```

```
this.osForm.controls.horaentrada.setValue(data.toLocaleTimeString());

this.osForm.controls.dataehoraentrada.setValue(dataEHoraEntrada.toISOStrin
g());
            }
          );
      } else {
        // instruções para execução no navegador
      }
    });
  }
}
```

Precisamos testar nossa aplicação, mas precisamos realizar algumas implementações no que já temos e aponto isso a seguir.

1. No arquivo `ordensdeservico-listagem.page.html` : em nosso `<ion-fab-button>` , vamos inserir dentro da tag o código `routerDirection="forward" [routerLink]="['/ordensdeservico-add-edit/-1']"` .

2. No `app-routing.module.ts` , precisamos adaptar a rota criada na geração dos códigos para que possa receber o argumento que estamos enviando. Veja o código na sequência.

```
{
    path: 'ordensdeservico-add-edit/:id',
    loadChildren: () => import('./pages/ordensdeservico/ordensdeservico-
add-edit/ordensdeservico-add-edit.module').then( m =>
m.OrdensdeservicoAddEditPageModule)
}
```

Execute agora sua aplicação e compare com a figura a seguir. Ela traz a visão de inserção, acessível a partir da visão de listagens de ordens de serviços, pela interação com o botão de adicionar em nosso FAB. A figura já traz o `select` em execução, apresentando uma listagem de clientes, que, no caso da figura, tem só um. Será

preciso, em `ordensdeservico-add-edit.module.ts`, inserirmos `ReactiveFormsModule` ao `imports` de `@NgModule`.

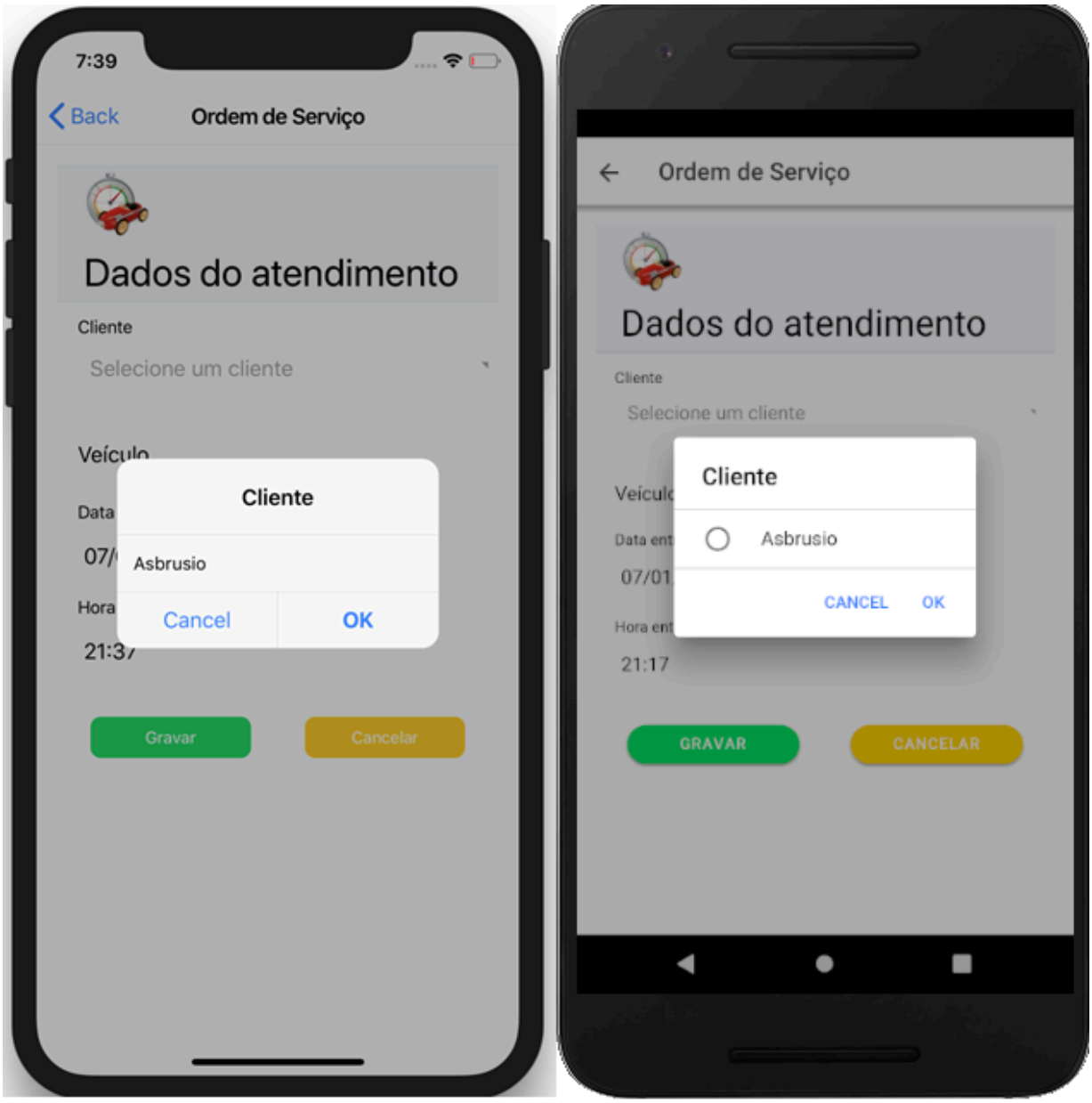


Figura 6.2: Página com a interface para inserção de ordens de serviço

6.5 Acesso a um atendimento existente

Nós temos a página de inserção de ordens de serviço pronta, mas nós ainda não registramos a alteração de uma ordem de serviço. Antes, quero que seja possível exibir a mesma visão para uma ordem de serviço nova, como vimos na implementação e teste anteriores. Precisamos realizar algumas alterações em nosso código. A primeira é em nossa listagem, no `<ion-item>`, dentro do `<ion-item-sliding>`. Precisamos incluir a URL de destino para o momento em que o usuário interagir com um item da listagem. Veja a alteração no código a seguir, no `ordensdeservico-listagem.page.html`. Observe como a URL foi particionada.

```
<ion-item routerDirection="forward" [routerLink]="['/', 'ordensdeservico-add-edit', ordemDeServico.ordemdeservicoid]">>
```

Com o identificador da ordem de serviço sendo enviado para nossa rota, nosso código para o template deverá estar preparado para recebê-lo e então recuperar a ordem de serviço da base de dados. Para isso, precisamos implementar um novo método em nosso `ordemdeservico.service.ts`, tal qual podemos ver na listagem a seguir.

```
public async getById(id: string): Promise<any> {
    try {
        const db = await this.databaseService.sqliteConnection.retrieveConnection(databaseName);
        const sql = 'select * from ordensdeservico where ordemdeservicoid = ?';
        try {
            db.open();
            const data = await db.query(sql, [id]);
            db.close();
            if (data.values.length > 0) {
                const ordemdeservico: OrdemDeServico = data.values[0];
                ordemdeservico.dataehoraentrada = new Date(ordemdeservico.dataehoraentrada);
                return ordemdeservico;
            } else {
                return null;
            }
        } catch (e) {
```

```
            return console.error(e);
        }
    } catch (e) {
        return console.error(e);
    }
}
```

O código anterior, em relação ao `getById()`, traz a cláusula `where` parametrizada, e este parâmetro é enviado ao `db.query()`, que é precedido pelo `db.open()` e sucedido pelo `db.close()`, que são, respectivamente, abertura e fechamento da conexão. Veja também que, após a verificação de recuperação do registro desejado, obtemos o primeiro item da coleção (índice 0). Como nossa propriedade `dataehoraentrada` é do tipo `Date`, precisamos converter o dado para este tipo antes de retornar o objeto recuperado.

Precisamos agora implementar a invocação dele em nosso código para o template de adição/edição. Para isso, vamos adaptar o `ionViewWillEnter()`, que já implementamos para a inserção, para o código que vemos na sequência. Estaremos aproveitando a implementação para torná-la funcional, tanto para inserção, como visualização dos dados da ordem de serviço selecionada na listagem. Essa implementação nova é no `ordensdeservico-add-edit.page.ts`, porém, antes, precisamos injetar no construtor as instruções a seguir.

```
private route: ActivatedRoute,
private ordensDeServicoService: OrdensDeServicoService,
```

Agora sim, bora pro código.

```
async ionViewWillEnter() {
    const id = this.route.snapshot.paramMap.get('id');

    const clientes = await this.clientesService.getAll();
    this.clientes = clientes;

    const isIdEmptyGUID = Guid.parse(id).isEmpty();
    const isIdValidGUID = Guid.isGuid(id);
```

```
    if (id && !isIdEmptyGUID && isIdValidGUID) {
        this.ordemDeServico = await
this.ordensDeServicoService.getById(id);
    } else {
        this.ordemDeServico = {ordemdeservicoid:
Guid.createEmpty().toString(),
            clienteid: Guid.createEmpty().toString(), veiculo: '',
dataehoraentrada: new Date() };
        this.modoDeEdicao = true;
    }

    this.osForm = this.formBuilder.group({
        ordemdeservicoid: [this.ordemDeServico.ordemdeservicoid],
        clienteid: [this.ordemDeServico.clienteid, Validators.required],
        veiculo: [this.ordemDeServico.veiculo, Validators.required],
        dataentrada: [{ value:
this.ordemDeServico.dataehoraentrada.toLocaleDateString(), disabled:
!this.modoDeEdicao }, Validators.required],
        horaentrada: [{ value:
this.ordemDeServico.dataehoraentrada.toLocaleTimeString(), disabled:
!this.modoDeEdicao }, Validators.required],
        dataehoraentrada: [this.ordemDeServico.dataehoraentrada]
    });

    this.ordemDeServico = this.osForm.value;
}
```

Notou que recuperamos o `id` fazendo uso de rotas? Por isso, precisamos injetar no construtor a propriedade `private route: ActivatedRoute` para isso. Depois, recuperamos os dados dos clientes registrados, pois precisaremos ter a coleção disponibilizada, tanto para selecionar um novo cliente, se for o caso, como para exibir o nome do cliente registrado para o atendimento.

Em seguida, verificamos se o `id` recebido refere-se a um já existente, para então recuperarmos seus respectivos dados da base com nosso novo método `getById()`. Caso o `id` não seja um já existente, é porque o usuário decidiu registrar um novo. O registro

do formulário é o mesmo que vimos para a inserção. Para a invocação deste método, precisamos injetar também no construtor a propriedade `private ordensDeServicoService: OrdensDeServicoService .`

Observe ainda que, como nosso objeto que será exibido na visão é diferente daquele que recuperamos e com o qual populamos nosso `osForm` , precisamos trazer para o `this.ordemDeServico` o objeto que será utilizado por nosso template HTML. Agora podemos testar nossa aplicação, tanto para inserção como para seleção de um atendimento já registrado.

6.6 Gravação dos dados do atendimento

Temos uma única página para manipulação das ordens de serviço, que atende tanto à inserção, quanto à atualização dos dados. Você deve ter notado que, quando executamos a inserção, o botão para gravação está habilitado. Entretanto, quando selecionamos um atendimento na listagem, os dados aparecem desabilitados e temos um botão com o texto `Alterar dados` , que habilitará os controles para atualização, caso esta seja a necessidade do usuário.

Com esta contextualização, podemos identificar dois pontos que precisamos implementar: a liberação para alteração dos dados e a gravação deles, que são atividades respectivas aos botões comentados anteriormente e, aproveitaremos para implementar também o que cancela a edição, retornando os dados à sua última versão gravada.

Entretanto, antes de partirmos para estas implementações, vamos abstrair o comportamento da gravação. Precisaremos atualizar o registro na base de dados, ou inserir o novo. Isso nos leva a implementar um serviço em nosso `ordensdeservico.service.ts` , que vamos chamar de `update()` , seguindo a mesma nomenclatura que adotamos para peças. Veja o código a seguir.

```
async update(ordemdeservico: OrdemDeServico): Promise<void> {
    let sql: any;
    let params: any;

    if (Guid.parse(ordemdeservico.ordemdeservicoid).isEmpty()) {
        ordemdeservico.ordemdeservicoid = Guid.create().toString();
        sql = 'INSERT INTO ordensdeservico(ordemdeservicoid, clienteid,
veiculo, dataehoraentrada) ' +
            'values(?, ?, ?, ?)';
        params = [ordemdeservico.ordemdeservicoid,
ordemdeservico.clienteid,
        ordemdeservico.veiculo, ordemdeservico.dataehoraentrada];
    } else {
        sql = 'UPDATE ordensdeservico SET clienteid = ?, veiculo = ?, ' +
            'dataehoraentrada = ? WHERE ordemdeservicoid = ?';
        params = [ordemdeservico.clienteid,
        ordemdeservico.veiculo, ordemdeservico.dataehoraentrada,
ordemdeservico.ordemdeservicoid];
    }

    try {
        const db = await
this.databaseService.sqliteConnection.retrieveConnection(databaseName);
        db.open();
        await db.run(sql, params);
        db.close();
    } catch (e) {
        console.error(e);
    }
}
```

Começamos declarando duas variáveis que serão responsáveis por conter a instrução SQL que será executada e os parâmetros que deverão ser utilizados nessa instrução. Em seguida, com base no valor de `ordemdeservico.ordemdeservicoid`, caso seja um GUID, verificamos que instrução SQL devemos preparar para a execução. Veja que, no caso de inserção, criamos um GUID para a propriedade. Após esta definição, executamos o método por meio da chamada à `db.run()`.

Com o serviço devidamente implementado, vamos retomar a linha de pensamento que nos levou a ele, que se refere à implementação dos métodos de gravação, cancelamento de edição e liberação da edição. Veja os métodos que precisaremos implementar no arquivo `ordensdeservico-add-edit.page.ts` na sequência.

```
// Método a ser invocado quando o botão de alterar dados for selecionado
iniciarEdicao() {
    this.modoDeEdicao = true;
}

// Método a ser invocado quando o botão de cancelar alteração for
selecionado
cancelarEdicao() {
    this.osForm.setValue(this.ordemDeServico);
    this.modoDeEdicao = false;
}

// Método a ser invocado quando o botão de gravar for selecionado
async submit() {
// Validação dos dados informados no formulário. Já trabalhamos com isso.
    if (this.osForm.invalid || this.osForm.pending) {
        await this.alertService.presentAlert('Falha', 'Gravação não foi
executada',
            'Verifique os dados informados para o atendimento', ['Ok']);
        return;
    }

// Aqui extraímos a data e hora da informadas no formulário e convertemos
para um Date
    const dataString = new
Date(this.osForm.controls.dataentrada.value).toDateString();
    const horaString = new
Date(this.osForm.controls.horaentrada.value).toTimeString();
    const dataEHora = new Date(dataString + ' ' + horaString);

// Invocamos o serviço, enviando um objeto com os dados recebidos da visão
    await this.ordensDeServicoService.update(
        {
```

```
            ordemdeservicoid: this.osForm.controls.ordemdeservicoid.value,
            clienteid: this.osForm.controls.clienteid.value,
            veiculo: this.osForm.controls.veiculo.value,
            dataehoraentrada: dataEHora,
        }
    );

// Informamos o usuário do sucesso da operação e o redirecionamos para a
listagem
    this.toastService.presentToast('Gravação bem sucedida', 3000, 'top');
    this.router.navigateByUrl('ordensdeservico-listagem');
}
```

Notou o uso da variável `modoDeEdicao` como responsável para definição do estado atual da aplicação em relação ao atendimento sendo exibido ao usuário?

Para a correta compilação do código anterior, precisamos injetar alguns objetos no construtor, que são: `private toastService: ToastService`, `private alertService: AlertService` e `private router: Router`.

Veja que estamos injetando um `AlertService`, que ainda não temos, mas trago na sequência o código que deverá ser implementado em um arquivo chamado `alert.service.ts`, na pasta `services`.

```
import { Injectable } from '@angular/core';
import { AlertController } from '@ionic/angular';

@Injectable({
    providedIn: 'root'
  })
export class AlertService {
    constructor(private alertCtrl: AlertController) {}

    async presentAlert(header: string, subHeader: string, message: string,
buttons: string[]) {
        const alert = await this.alertCtrl.create({
          header,
```

```
          subHeader,
          message,
          buttons
        });
        await alert.present();
    }
}
```

No processo de redirecionamento, poderíamos pensar que, se estivéssemos em uma inserção, seria interessante ficar na mesma página, para que uma nova pudesse ocorrer, mas isso fica com você, já temos a lógica e ferramental para isso.

Ainda não está pronto. Temos os métodos implementados, mas precisamos atribuí-los em nosso template, nos controles, para que o efeito desejado pelo usuário na interação com os botões possa ocorrer. No botão de alterar dados, insira `(click)="iniciarEdicao()"` e, no de cancelar, insira `(click)="cancelarEdicao()"`.

Para a gravação, precisamos realizar duas alterações. A primeira é na definição do `<form>`, onde devemos inserir `(submit)="submit()"`, que liga nosso método ao processo de submissão do formulário. A segunda, agora sim, no botão, é apenas inserir `type="submit"` no código da tag do botão de gravação.

Agora podemos testar nossa aplicação para inserirmos um novo atendimento e selecionarmos um que seja exibido na listagem de atendimentos registrados.

6.7 Removendo um atendimento já registrado

Falta-nos a remoção de um atendimento, que é o que faremos nesta seção. Sabemos que é preciso um método em nosso arquivo `ordensdeservico.service.ts` e podemos vê-lo na listagem a seguir.

```
async removeById(id: string): Promise<boolean | void> {
    try {
        const db = await
this.databaseService.sqliteConnection.retrieveConnection(databaseName);
        db.open();
        await db.run('DELETE FROM ordensdeservico WHERE ordemdeservicoid =
?', [id]);
        db.close();
        return true;
    } catch (e) {
        console.error(e);
    }
}
```

Veja que no `generics` para o tipo de retorno estamos informando dois tipos de dados, o `boolean`, que é para o sucesso, e `void`, para o caso de exceção.

Com o serviço pronto, precisamos implementar a invocação a ele em nosso arquivo `ordensdeservico-listagem.page.ts`, pois a exclusão será possível por meio da listagem, com a interação em menu que aparecerá quando o usuário deslizar um item. Veja esse método na sequência e note que precisamos injetar o `AlertService` e o `ToastService`, caso já não estejam injetados.

```
async removerAtendimento(ordemdeservico: OrdemDeServico) {
    await
this.ordensdeservicoService.removeById(ordemdeservico.ordemdeservicoid)
        .then( async () => {
            this.ordensDeServico = await
this.ordensdeservicoService.getAll();
            this.toastService.presentToast('Ordem de Serviço removida',
3000, 'top');
            await this.slidingList.closeSlidingItems();
        })
        .catch( async (e) =>  await
this.alertService.presentAlert('Falha', 'Remoção não foi executada', e,
['Ok']));
}
```

Agora, em nosso template, precisamos inserir o código para exibir o botão de remoção e, neste código, temos a requisição ao método que implementamos anteriormente.

```
<ion-item-options side="end">
    <ion-item-option color="danger"
(click)="removerAtendimento(ordemDeServico)">
      <ion-button color="danger" no-padding>
          <ion-icon name="trash"></ion-icon>
          Remover
      </ion-button>
    </ion-item-option>
</ion-item-options>
```

Podemos testar nossa aplicação e remover um atendimento de nossa listagem. Não apresentarei figuras, pois é semelhante ao que já vimos.

Conclusão

Terminamos este interessante capítulo. Não vimos muitos recursos do Ionic nele, mas nos dedicamos a técnicas e serviços, e concluímos um CRUD com uma base de dados local na aplicação, o SQLite. Infelizmente o processo de `livereload` com o plugin disponibilizado não é muito produtivo. Em nosso próximo capítulo, faremos uso do Firestore, que, de maneira resumida, é uma base de dados com base em documentos, disponibilizada remotamente, na nuvem. Também implementaremos uma página específica para pesquisa, que substituirá nosso `<ion-select>`. Será um capítulo muito legal. Já nos vemos!

CAPÍTULO 7
CRUD com Firestore para acesso a dados remoto e uma página de pesquisa

No capítulo anterior adaptamos nossa aplicação para uma "cara" mais próxima de um aplicativo móvel, com o Sidemenu e manipulação de dados permanentes, em uma base de dados SQLite. Vimos o componente `<ion-select>`, que nos apresentou uma coleção de dados, onde elegemos um para compor os dados de um atendimento a um veículo.

Entretanto, as aplicações móveis, em relação aos dados manuseados por ela, fazem uso cada vez mais frequente e restrito dos dados remotos, inclusive, em alguns casos, sem a necessidade de sincronização com uma base local. E é isso que trabalharemos neste capítulo, por meio do Firestore, que apresentarei mais adiante com definições e características.

Em relação ao Firestore, é importante ficar claro que a cada mudança no modelo de negócio, a estrutura da base também muda. Fique atento(a) a isso em seus testes.

Com exceção do acesso ao Firestore e das instruções para manipulação dos dados, nossas implementações neste capítulo trabalharão com técnicas semelhantes às que estamos vendo desde o capítulo 5. Mas veremos também uma solução mais indicada para seleção de um dado para utilização em um formulário, em substituição ao `<ion-select>`. Criaremos uma página específica para pesquisa de dados.

7.1 Preparação do projeto para o Firestore

De acordo com Jorge Vergara (https://blog.ionicframework.com/building-ionic-apps-with-firestore/), o Firestore é um banco de dados baseado em documentos NoSQL, totalmente gerenciado para dispositivos móveis e web. Ele é projetado para armazenar e sincronizar dados de aplicativos com facilidade. Ou seja, é um produto que lida diretamente com aplicações para dispositivos móveis, sem necessidades de nos preocuparmos com camadas além da nossa aplicação.

O Firestore teve sua origem no Firebase, ambos produtos do Google e, caso queira, você pode obter um texto específico para a tecnologia em https://firebase.google.com/docs/firestore/.

Falamos anteriormente sobre documentos NoSQL e que o Firestore é uma base de dados que tem como estrutura estes documentos, esse paradigma tende a ser mais específico e mais simples.

Mas o que é NoSQL? O que são documentos? O NoSQL trata as tabelas como coleções e seus registros como documentos, que são objetos baseados em entradas específicas a um domínio de dados. Uma grande diferença que surpreende os iniciantes é que uma base relacional, como a que vimos nos capítulos anteriores, precisa de estruturas predefinidas para ser populada. Se um novo dado surgir, precisamos alterar a estrutura das tabelas, para receber este dado. Com NoSQL, essa preocupação deixa de existir, é dinâmica, cada objeto da coleção tem sua estrutura atrelada aos dados que o compõem. Os tipos de dados possíveis para cada objeto são muitos, podendo inclusive uma coleção fazer parte de outra.

Instalação dos plugins necessários

Após ter decidido por criar um novo projeto ou trabalhar no capítulo anterior, acesse o terminal, na pasta do projeto, e execute a instrução a seguir para a instalação do firebase e suas ferramentas.

```
npm install -g firebase-tools
```

Execute `firebase --version` em seu terminal. Se tudo estiver OK, a versão instalada deve ser exibida a você. É preciso ser no mínimo a `9.23.3`.

Agora vamos instalar os componentes que serão consumidos pelos artefatos de nossa aplicação. Execute a instrução a seguir no terminal. O `rxfire` normalmente é instalado em conjunto com o firebase, porém, em alguns casos, ocorre um erro, então já coloquei ele para ser instalado de maneira forçada.

```
npm install @angular/fire firebase rxfire
```

Com os plugins e ferramentas devidamente instalados, precisamos acessar o portal do `Firebase Console`, por meio do link https://console.firebase.google.com. Por ser um produto Google, é importante que tenhamos uma conta do Gmail para realizarmos o acesso à plataforma.

Com o acesso devidamente realizado, o serviço oferece a opção de `Adicionar Projeto`. Forneça um nome para seu projeto, a ferramenta verificará a disponibilidade para ele. Configure as localizações solicitadas de acordo com sua necessidade. Eu optei por não ativar o `Google Analytics`. Ao final da interface, confirme o aceite para os termos de acesso e a criação do projeto. Você será informado quando a criação for concluída.

A página que será exibida após a confirmação da mensagem de conclusão da criação do projeto é conhecida por `Get Started` do `Project Overview`, que tem um link de acesso na barra lateral do `Firebase Console`. Esta página oferece alguns botões relacionados às plataformas, `iOS`, `Android`, `Web </>` e `Unit`. Nós confirmaremos a criação para uma aplicação web, lembrando que nossas aplicações Ionic são executadas em um ambiente web, por padrão. Ao iniciar a criação, forneça o nome para ela, eu não confirmei a criação do `Firebase Hosting` e cliquei no `Registrar App`.

Em uma nova página, serão exibidas instruções, como as exibidas pelo `npm install firebase`, que já executamos. Também será

informado um conjunto de código JavaScript, que deve ser copiado, pois usaremos essas informações em nosso código. Para efeito ilustrativo, este código é semelhante ao apresentado na sequência. Copie e guarde este código, pois já o utilizaremos.

Como nossa aplicação não é uma aplicação web comum, não podemos copiar o código apresentado no arquivo HTML, como sugere a janela comentada anteriormente. Precisamos, em nosso projeto Ionic, na pasta `/src/app`, criar o arquivo `credentials.ts` e usar parte do código anterior atrelado à sintaxe do JavaScript. Veja este código na sequência. Com o `export`, poderemos importar este objeto em qualquer ponto da nossa aplicação.

```
export const environment = {
    production: false,
    useEmulators: true,
    firebaseConfig: {
        apiKey: "AIzaSyDmlXzNpTayhN4jCWDkgAi-qFWvxtMSKgs",
        authDomain: "ionic5-c0dc4.firebaseapp.com",
        databaseURL: "https://ioniccgfb.firebaseio.com",
        projectId: "ionic5-c0dg4",
        storageBucket: "ionic5-c0dg4.appspot.com",
        messagingSenderId: "624846692813",
        appId: "1:624846692713:web:40d3e49ffdeb3940939gf8"
    },
};
```

Com nossa configuração implementada no projeto, precisamos iniciar o Firebase em nosso já conhecido arquivo `app.module.ts`, começando com a importação dos pacotes `AngularFire2` que instalamos anteriormente. Veja as instruções que devemos adicionar no arquivo a seguir.

```
import { provideFirebaseApp, initializeApp } from '@angular/fire/app';
import { enableIndexedDbPersistence, getFirestore, provideFirestore } from
'@angular/fire/firestore';
import { environment } from './credentials';
```

Precisamos importar para nosso módulo os módulos referenciados anteriormente e fazemos isso inserindo as instruções a seguir no `imports` dentro de nosso `@NgModule()`. Veja que, ao chamarmos o `AngularFireModule.initializeApp(firebaseConfig)`, estamos enviando nossas credenciais para a inicialização da aplicação, o que fará com que nossa aplicação conecte-se com o `Firebase`.

```
provideFirebaseApp(() => initializeApp(environment.firebaseConfig)),
```

Para possibilitar o uso de emuladores, após a linha anterior, implemente o código a seguir:

```
provideFirestore(() => {
  if (environment.useEmulators) {
    const firestore = getFirestore();
    enableIndexedDbPersistence(firestore);
    return firestore;
  } else {
    getFirestore();
  }
}),
```

7.2 Populando a base de dados do Firestore

No capítulo anterior, nós trabalhamos um CRUD para `Ordens de Serviço`, onde fazíamos uso de uma associação/relacionamento com `Clientes`, mas tínhamos dados previamente inseridos na tabela. Agora vamos implementar o CRUD para `Clientes` e, diferente das implementações anteriores, para podermos variar um pouco, vamos começar pela inserção de clientes. Já sabemos que temos uma série de burocracias para termos este serviço disponibilizado em nossa aplicação e faremos isso nas subseções que se seguem, começando pela execução do código da sequênciar na pasta de seu projeto, no terminal, para a criação dos artefatos que precisaremos para inserção e alteração de clientes.

```
ionic generate page pages/clientes/clientes-add-edit
```

O template para a página de registro

Em `/src/app/pages`, temos agora a pasta `clientes` e, nela, outra, chamada `clientes-add-edit`. Com esta estrutura básica que já conhecemos e também como já sabemos precisamos realizar adaptações no arquivo `clientes-add-edit.page.html` de acordo com o que pode ser visto a seguir.

```
<ion-header>
    <ion-toolbar>
        <ion-buttons slot="start">
            <ion-back-button defaultHref="clientes-listagem"></ion-back-
button>
        </ion-buttons>
        <ion-title>Clientes</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content padding>
    <ion-item-divider color="light">
        <ion-grid no-padding>
            <ion-row no-padding>
                <ion-col col-3 align-self-center>
                    <ion-img style="width: 50px; height: 50px;"
src="assets/imgs/icon_clientes.png"></ion-img>
                </ion-col>
                <ion-col col-9>
                    <ion-label style="font-size: 30px">Dados do
cliente</ion-label>
                </ion-col>
            </ion-row>
        </ion-grid>
    </ion-item-divider>

    <ng-container *ngIf="clienteForm">
        <form [formGroup]="clienteForm">
            <ion-item lines="none" [disabled]="!modoDeEdicao">
```

```
                <ion-label position="floating" class="label-input-
title">Nome</ion-label>
                <ion-input formControlName="nome" type="text"
style="margin-left: 3%;"></ion-input>
            </ion-item>

            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-label position="floating" class="label-input-
title">Email</ion-label>
                <ion-input formControlName="email" type="email"
style="margin-left: 3%;"></ion-input>
            </ion-item>

            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-label position="floating" class="label-input-
title">Telefone</ion-label>
                <ion-input formControlName="telefone" type="tel"
style="margin-left: 3%;"></ion-input>
            </ion-item>

            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-label position="floating" class="label-input-
title">Renda</ion-label>
                <ion-input formControlName="renda" type="number"
style="margin-left: 3%;"></ion-input>
            </ion-item>

            <div (click)="selecionarData()">
                <ion-grid class="ion-no-padding">
                  <ion-row>
                    <ion-col size="9">
                      <ion-item lines="none">
                        <ion-label position="stacked">Nascimento</ion-
label>
                        <ion-input formControlName="nascimentoForm"></ion-
input>
                      </ion-item>
                    </ion-col>
                    <ion-col class="ion-no-padding">
```

```
                    <ion-item lines="none">
                      <ion-button fill="clear" class="ion-no-margin;ion-
no-padding">
                        <ion-icon icon="calendar"></ion-icon>
                      </ion-button>
                    </ion-item>
                  </ion-col>
                </ion-row>
              </ion-grid>
            </div>

            <ion-button *ngIf="!modoDeEdicao" shape="round"
color="primary" expand="block" padding>
                Alterar dados
            </ion-button>
            <div *ngIf="modoDeEdicao" no-padding>
                <ion-grid>
                    <ion-row>
                        <ion-col col-6 >
                            <ion-button shape="round" color="success"
size="small" padding expand="block">
                                Gravar
                            </ion-button>
                        </ion-col>

                        <ion-col col-6>
                            <ion-button shape="round" color="warning"
size="small" padding expand="block">
                                Cancelar
                            </ion-button>
                        </ion-col>
                    </ion-row>
                </ion-grid>
            </div>
        </form>
    </ng-container>
</ion-content>
```

A classe TypeScript para nosso template

Vamos então para a implementação de nossa classe, que, como já sabemos, é responsável pela renderização de nosso template HTML. Veja o código dela logo na sequência. Adapte a que foi criada na geração de nosso artefato. De início, o objeto `cliente` está inicializado como novo, pois nosso objetivo agora é a criação de um novo cliente. Precisaremos trazer nosso arquivo `cliente.model.ts` para nosso projeto, caso você tenha criado um projeto do início.

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormBuilder, Validators } from '@angular/forms';
import { Cliente } from 'src/app/models/cliente.model';
import { MenuController } from '@ionic/angular';

@Component({
    templateUrl: './clientes-add-edit.page.html'
})
export class ClientesAddEditPage implements OnInit {
    private cliente: Cliente;
    public modoDeEdicao = false;
    public clienteForm: FormGroup;

    constructor(
        private formBuilder: FormBuilder
    ) {}

    async ngOnInit() {
    }

    async ionViewWillEnter() {
        this.modoDeEdicao = true;

        this.cliente = {clienteid: '', nome: '', email: '', telefone: '',
renda: 0.00, nascimento: new Date()};

        this.clienteForm = this.formBuilder.group({
            clienteid: [this.cliente.clienteid],
            nome: [this.cliente.nome, Validators.required],
            email: [this.cliente.email, Validators.required],
            telefone: [this.cliente.telefone, Validators.required],
```

```
            renda: [this.cliente.renda, Validators.required],
            nascimentoForm: [{ value:
this.cliente.nascimento.toLocaleDateString(), disabled:
!this.modoDeEdicao}, Validators.required],
            nascimento: ['', Validators.required]
        });
    }

    selecionarData() {
        if (!this.modoDeEdicao) {
          return;
        }

        this.platform.ready().then(() => {
          if (this.platform.is('capacitor')) {
            this.datePicker.show({
              date: new Date(),
              mode: 'date',
              locale: 'pt-br',
              doneButtonLabel: 'Confirmar',
              cancelButtonLabel: 'Cancelar'
            })
              .then(
                data => {

this.clienteForm.controls.nascimentoForm.setValue(data.toLocaleDateString(
));

this.clienteForm.controls.nascimento.setValue(data.toISOString());
                }
              );
          } else {
            // instruções para execução no navegador
          }
      });
}
```

A classe para o módulo de manutenção dos dados do cliente

Pois bem. Vamos à etapa final, que se refere à classe em que definimos o módulo. Neste arquivo, basta adicionarmos em `imports`, o `ReactiveFormsModule`.

Já temos a interface com o usuário pronta. Agora precisamos testar nossa aplicação. Só que, para isso, tal qual fizemos para os atendimentos no capítulo anterior, precisamos realizar algumas implementações no que já temos.

Vamos inserir o código a seguir como um novo elemento da variável `pages` no arquivo `app.component.ts`. Aqui também teremos que trazer o nosso `sidemenu`, caso você tenha criado um projeto do início. Você poderá comentar os códigos para atendimentos ou já importar também tudo o que for necessário.

```
,
{
  title: 'Clientes',
  url: '/clientes-add-edit',
  icon: '/assets/imgs/icon_clientes.png'
}
```

Como não utilizaremos a persistência local neste capítulo, já podemos tirar o código responsável por isso e que está no `app.component.ts`, que é o corpo do construtor e do `ngOnInit()`, mas lembre-se que o que temos relacionado à ordens de serviço deixará de funcionar.

Execute a aplicação e compare com as figuras a seguir. Elas trazem a visão de inserção, mas não temos ainda nenhum comportamento em relação à interação com o usuário. Você verá também no Sidemenu a nova opção para clientes, basta clicar nela.

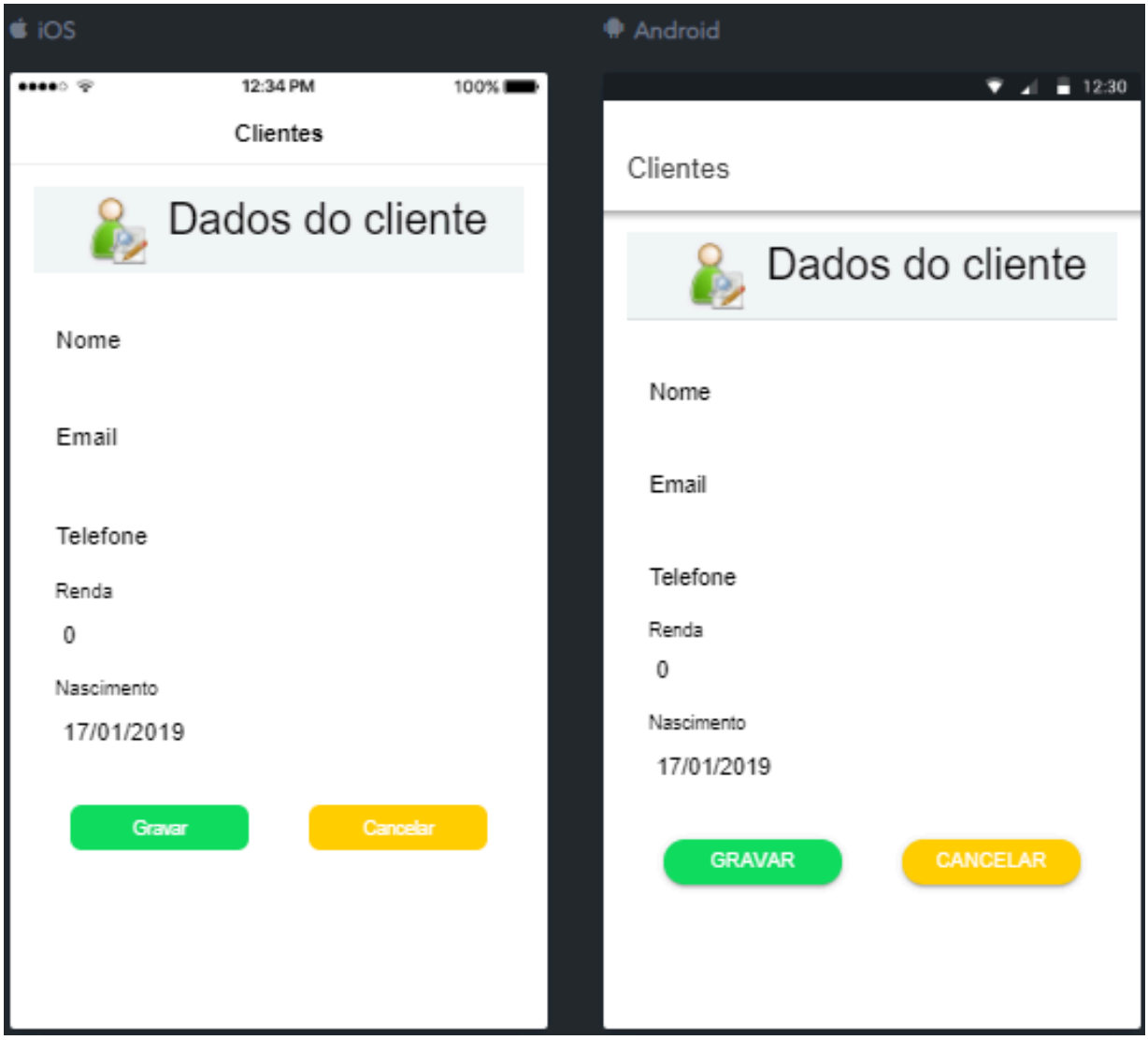


Figura 7.1: Página com a interface para inserção de clientes

Gravação dos dados do cliente

Seguiremos para clientes a mesma metodologia dos exemplos anteriores. Teremos uma única visão para visualização dos dados, alteração e inserção. Desta maneira, as implementações trabalhadas aqui já terão como foco esta característica.

Como estamos trabalhando a situação de criação de um novo cliente, precisaríamos nos preocupar com a criação do valor para a propriedade `clienteid`, que é identificadora de cada objeto, porém o

`Firestore` resolve isso no momento da criação de um novo objeto em sua base.

Vamos para a implementação específica de um serviço para clientes. O primeiro código que trabalharemos é em nossa classe de serviço, do arquivo `clientes.service.ts`, que já temos implementado no capítulo anterior, no qual recuperamos os clientes e populamos nosso `<ion-select>`. Sugiro que, neste momento, você comente o código existente, pois agora mudamos nossa base de dados e a implementação também muda. Vamos então ao código para criação de um novo cliente. Veja a listagem a seguir, observando os importes para o Firestore.

```
import { Injectable } from '@angular/core';
import { Cliente, clienteConverter } from '../models/cliente.model';
import { Firestore, collection, getDocs, setDoc, doc, query, orderBy,
getDoc, deleteDoc } from '@angular/fire/firestore';


@Injectable({
    providedIn: 'root'
})
export class ClientesService {
    constructor(
        private _fireStore: Firestore,
    ) {
    }

    async create(cliente: Cliente): Promise<void> {
        try {
            cliente.nascimento = new Date(cliente.nascimento);

            const clientesRef = collection(this._fireStore, "clientes");

            await setDoc(doc(clientesRef), {
                nome: cliente.nome,
                email: cliente.email,
                telefone: cliente.telefone,
                renda: cliente.renda,
```

```
                nascimento: cliente.nascimento,
            });
        } catch (e) {
            console.error(e);
        }
    }
}
```

Na implementação de `create()` temos a chegada de um `Cliente` e o retorno padrão para o serviço que consumiremos, que é um `Promise<void>`. No bloco `try`, temos a obtenção de uma referência à coleção `clientes`, no `Firestore`. Na base de dados, ajustamos a data de nascimento do cliente para um `Date`, pois ele chega como `string`.

Na sequência, invocamos o método `setDoc()`, que recebe o retorno de `doc()`, que tem a referência à coleção como argumento. Neste contexto, uma coleção pode ser vista como uma tabela em bancos relacionais. Como segundo argumento, enviamos um Json, mapeado com dados do cliente recebido. Aqui poderíamos pensar em criar na classe de modelo um método que transforme o objeto em um Json, fica a dica.

Vamos para o método `submit()`, de nosso arquivo que renderiza o template de nossa página. Verifique, no código a seguir, a simplicidade de compreensão do código.

```
async submit() {
    if (this.clienteForm.invalid || this.clienteForm.pending) {
      await this.alertService.presentAlert('Falha', 'Gravação não foi
executada',
        'Verifique os dados informados para o atendimento', ['Ok']);
      return;
    }

    const loading = await this.loadingCtrl.create();
    await loading.present();

    if (this.cliente.clienteid === '') {
```

```
      await this.clientesService.create(this.clienteForm.value)
        .then(
          () => {
            loading.dismiss().then(() => {
              this.toastService.presentToast('Gravação bem sucedida',
3000, 'top');
                          this.router.navigateByUrl('/clientes-add-edit)');
            });
          },
          error => {
            console.error(error);
          }
        );
    }
}
```

No código anterior, caso o formulário seja inválido ou não esteja pronto, fazemos uso do `alertService` para informar o usuário. Com isso, precisamos injetar no construtor o código `private alertService: AlertService`.

Após a validação do formulário, temos o uso de um novo componente, um `LoadingController`, que devemos injetar também no construtor, com a instrução `private loadingCtrl: LoadingController`. Daremos ao usuário o sentimento de que a aplicação está processando uma requisição, evitando a impressão de congelamento de tela. Em seguida, verificamos se o objeto não tem um valor para `clienteid`, nossa chave/identificador e, neste caso, invocamos nosso serviço para criação do objeto na base, mas, precisamos inserir no construtor o código `private clientesService: ClientesService`.

Com a finalização bem-sucedida do `create()`, utilizamos o `router`, que também precisamos injetar em nosso construtor, com `private router: Router`. Antesdo `router`, fazemos também uso do `toastService`. Precisamos injtá-lo (`private toastService: ToastService`). A princípio, redirecionamos a aplicação para a mesma página, mas isso pode ser dinâmico, dependendo sempre da necessidade em nosso construtor, relacionada à operação concluída. Precisaremos

também da injeção `private toastService: ToastService` em nosso construtor.

Para a gravação do novo cliente funcionar, temos que realizar duas alterações. A primeira é na definição do `<form>`, onde devemos inserir `(submit)="submit()"`, que liga nosso método ao processo de submissão do formulário. A segunda, agora sim no botão, é apenas inserir `type="submit"` no código da tag do botão de gravação.

Antes de testarmos nossa aplicação, vamos completar nossa configuração do Firestore na `Console Firebase`. Na página da aplicação, temos um item de menu chamado `Firestore Database`, vamos acessá-lo. Ao confirmar o primeiro acesso à base de dados do Firebase, como ainda não temos nenhuma criada, um link para a criação dela será oferecido, então clique nele. Uma janela será exibida, solicitando as regras que devem ser implementadas para o acesso à base. A princípio, vamos liberar o acesso global, selecionando a opção `Iniciar modo de teste`, que não requer autenticação. É possível alterar esta configuração depois e recomendo que realmente o faça. Mais à frente, quando trabalharmos autenticação, retomaremos esse assunto. Confirme o processo clicando o link/botão `Ativar`. A página com a relação de nossas coleções será exibida, mas não temos nada ainda.

Antes de testarmos nossa aplicação, é preciso comentarmos o método `ionViewWillEnter()` de `ordensdeservico-add-edit.page.ts`, pois há nele uma invocação ao `getAll()` de `ClientesService`, que não ajustamos ainda ao `Firestore`.

Muito bem. Agora podemos testar nossa aplicação para inserirmos um novo cliente. Informe os dados e realize a gravação. Após isso, retorne ao Firebase Console e, em `Firestore Database`, veja o dado já registrado, tal qual podemos ver na figura a seguir.

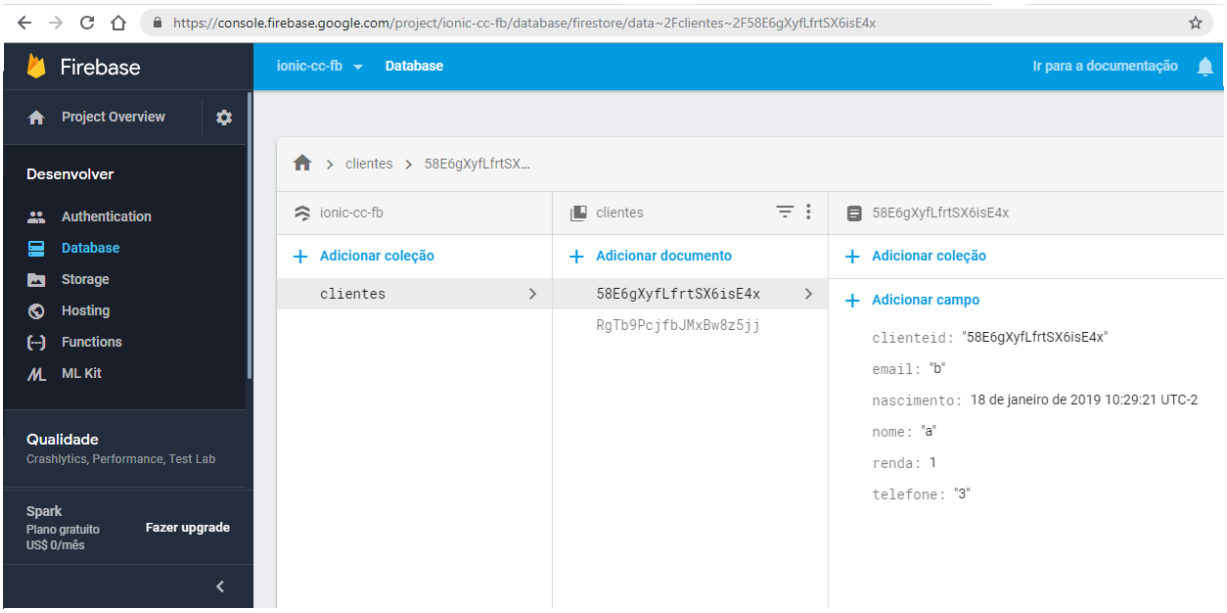


Figura 7.2: Visualização de dados inseridos no Firebase.

7.3 Visualização dos clientes registrados

Vamos agora para a implementação da página responsável pela exibição de todos os clientes que já temos persistidos em nossa base de dados remota, no Firestore. Trabalharemos em subseções que apontarão cada arquivo que criaremos ou adaptaremos para esta funcionalidade.

Método getAll() em ClientesService

No capítulo anterior, implementamos o método `getAll()` em `ClientesService`, pois precisávamos dos clientes no registro e manipulação dos dados referentes a atendimentos de clientes na aplicação, e funcionou perfeitamente para a tecnologia que adotamos para o momento, que era uma base de dados SQLite. Agora que estamos trabalhando com o `Firestore`, precisamos então adaptar este método e podemos vê-lo na sequência. Lembre-se de

que ele deverá estar em `clientes.service.ts`. Os imports necessários já foram realizados quando implementamos o `create()`.

```
async getAll(): Promise<Cliente[]> {
    const clientes: Cliente[] = [];
    const q = query(collection(this._fireStore, "clientes"),
orderBy("nome")).withConverter(clienteConverter);
    const querySnapshot = await getDocs(q);
    querySnapshot.forEach((doc) => {
        clientes.push(doc.data());
    });
    return clientes;
}
```

Observe que retornamos uma coleção de `Cliente`, mas você não vê neste código o mapeamento do Json que vem do Firebase para nosso modelo de negócio, mas é possível ver que estamos utilizando um conversor por meio do código `withConverter(clienteConverter)`, ao final da instrução que atribui um resultado a `q`. Mas onde temos este `clienteConverter`? Em nosso arquivo `cliente.model.ts`. Veja este código na sequência, que deve estar após a declaração da interface.

```
export const clienteConverter = {
    toFirestore: (cliente) => {
        return <Cliente>{
            nome: cliente.nome,
            email: cliente.email,
            telefone: cliente.telefone,
            renda: cliente.renda,
            nascimento: cliente.nascimento,
        };
    },
    fromFirestore: (snapshot, options) => {
        const data = snapshot.data(options);
        return <Cliente>{
            clienteid: snapshot.id,
            nome: data.nome,
            email: data.email,
            telefone: data.telefone,
```

```
            renda: data.renda,
            nascimento: data.nascimento.toDate(),
        }
    }
};
```

Agora sim, realize todas as importações necessárias ao `getAll()`. Para ajudar, o código está na sequência.

```
import { Cliente, clienteConverter } from '../models/cliente.model';
import { collection, doc, Firestore, getDocs, orderBy, query, setDoc }
from 'firebase/firestore';
```

Verificou no código, anterior aos imports, que já temos a implementação da conversão do modelo para Json? Poderemos utilizar isso no registro e atualização dos clientes, pois lá ainda temos o mapeamento, mas veremos isso depois.

Observe que, para o campo `nascimento`, convertemos o dado que vem do Firestore para uma formatação em data, por meio do `toDate()`. Isso se faz necessário pelo fato de o Firestore retornar uma data como `Timestamp`, o que causa erros em nossa aplicação, da maneira como estamos fazendo.

Caso você tenha optado por manter seu projeto tal qual tínhamos no capítulo anterior, certamente um erro de compilação ocorrerá com essa nova implementação na página de registro de ordens de serviço, pois a população dos clientes lá esperava um tipo de dado diferente deste que estamos adaptando.

Para contornarmos esta situação neste momento, você pode optar por comentar o código onde há o erro da compilação, até o momento de chegarmos nele, para registrar as adaptações necessárias.

O template para a página de listagem de clientes

Vamos agora criar os artefatos necessários para nossa página de listagem de clientes. Veja a instrução a seguir, a ser executada em

seu terminal.

```
ionic generate page pages/clientes/clientes-listagem
```

Vamos começar a adaptação nos códigos gerados, e a primeira será no arquivo chamado `clientes-listagem.page.html`, com o código que temos na sequência.

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
      <ion-back-button defaultHref="/dashboard"></ion-back-button>
    </ion-buttons>
    <ion-title>Clientes</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content padding-left padding-top>
  <ion-list #slidingList>
    <ion-item-sliding *ngFor="let cliente of clientes">
      <ion-item routerDirection="forward"
      [routerLink]="['/', 'clientes-add-edit', cliente.clienteid]">
        <ion-label>
          <h2>{{cliente.nome}}</h2>
          <p style="text-align:right">Telefone: {{cliente.telefone}}</p>
        </ion-label>
      </ion-item>

      <ion-item-options side="end">
        <ion-item-option color="danger" (click)="removerCliente(cliente)">
          <ion-button color="danger" no-padding>
            <ion-icon name="trash"></ion-icon>
            Remover
          </ion-button>
        </ion-item-option>
      </ion-item-options>
    </ion-item-sliding>
  </ion-list>

  <ion-fab vertical="top" horizontal="end" slot="fixed" edge="true">
```

```
    <ion-fab-button routerDirection="forward" [routerLink]="['/clientes-
add-edit/-1']">
      <ion-icon name="add"></ion-icon>
    </ion-fab-button>
  </ion-fab>
</ion-content>
```

A classe que fornecerá interação para o template de listagem

Vamos agora adaptar o arquivo `clientes-listagem.page.ts` com o código apresentado na sequência. Observe o comportamento para o método `ionViewWillEnter()`, que simplesmente invoca, de maneira assíncrona, o método que implementamos anteriormente e atribui à variável `clientes`, seu resultado.

```
import { Component, OnInit } from '@angular/core';
import { Cliente } from 'src/app/models/cliente.model';
import { ClientesService } from 'src/app/services/clientes.service';

@Component({
  selector: 'app-clientes-listagem',
  templateUrl: './clientes-listagem.page.html',
  styleUrls: ['./clientes-listagem.page.scss'],
})
export class ClientesListagemPage implements OnInit {

  public clientes: Cliente[];

  constructor(private clientesService: ClientesService) { }

  ngOnInit() {
  }

  async ionViewWillEnter() {
    this.clientes = await this.clientesService.getAll();
  }
}
```

Adaptações para testarmos a aplicação

Temos nossa página de listagem já implementada, precisamos agora adaptar nossa aplicação para que, em vez de exibir a página de inserção, exiba a de listagem, pois é a partir da listagem que chegaremos na de inserção.

Vamos alterar o código que temos para que, na `url`, tenhamos agora a listagem. Isso é no `app.component.ts`. Para ajudar, trago todo o trecho a ser modificado na sequência.

```
,
{
  title: 'Clientes',
  url: '/clientes-listagem',
  icon: '/assets/imgs/icon_clientes.png'
},
```

Após a gravação do cliente, precisamos redirecionar nosso usuário para a página de listagem. Desta maneira, ajuste a instrução a seguir, que já existe abaixo do toast da confirmação de gravação.

```
this.router.navigateByUrl('/clientes-listagem');
```

Execute a aplicação e verifique que a listagem é exibida já com o registro que criamos na implementação anterior.

O Firebase oferece meios para que seja mantida uma ligação direta dos dados recuperados em sua aplicação com os dados na base remota, o que possibilita que esses dados sejam sincronizados de acordo com as atualizações realizadas em qualquer uma das pontas, mas, como o objetivo aqui é o Ionic e não recursos do Firebase, deixarei isso para você. Há bastante documentação sobre tudo isso em https://firebase.google.com/docs/firestore/query-data/listen?hl=pt-br.

7.4 Consulta e alteração de um cliente

Vamos trabalhar agora a página de nossa aplicação que exibirá os dados de um cliente selecionado na listagem. Não temos nada de novo em relação à técnica que vínhamos implementando. O que teremos de novidade está ligado diretamente a como nosso serviço será implementado, pois estamos utilizando uma tecnologia nova para nós. Mas veremos que é tranquilo e poderemos discutir alguns pontos.

O método getById() em ClientesService

Tal qual trabalhamos para a listagem, vamos começar as especificidades dessa nova funcionalidade, apresentando a implementação do nosso serviço que realizará a busca do cliente com base no ID do cliente selecionado. Este envio já está definido na rota que implementamos nas seções anteriores, referentes à listagem. Veja o método a seguir.

```
async getById(clienteId: string): Promise<Cliente> {
    const q = doc(this._fireStore, "clientes",
clienteId).withConverter(clienteConverter);
    const querySnapshot = await getDoc(q);

    return querySnapshot.data();
}
```

Veja o retorno do método anterior, uma `Promise`, o que nos permitirá trabalhar com `async` e `await`.

O foco para este serviço que estamos implementando é termos o objeto recuperado, para que possamos manter a base do que estamos utilizando com formulários nas páginas. Temos o `doc()` retornando um objeto que utilizamos para enviar como argumento à invocação de `getDoc()`, que é assíncrona.

Quando a invocação comentada anteriormente for completada, já retornamos o dado para o método invocador. Poderíamos trabalhar

aqui exceções e verificação de existência ou não do documento desejado, mas deixarei esta etapa com você.

Adaptações para a classe do template

Nós já temos nossa página de inserção de clientes pronta e funcionando e, para o modelo de aplicação que estamos utilizando, precisamos, de início, realizar apenas algumas modificações em nossa classe para o template. Veja a nova implementação para o `ionViewWillEnter()` que está na sequência. Verifique se injetou no construtor o `private route: ActivatedRoute` .

Observe que aplicamos ao código a lógica necessária para verificar se estamos inserindo um novo cliente ou visualizando os dados de um que tenha sido selecionado na listagem. Conseguiu notar a similaridade com a lógica que aplicamos nos capítulos anteriores? Viu que a mudança está praticamente no serviço implementado anteriormente?

```
async ionViewWillEnter() {
    const id = this.route.snapshot.paramMap.get('id');

    if (id !== '-1') {
      this.cliente = await this.clientesService.getById(id);
    } else {
      this.cliente = { clienteid: '', nome: '', email: '', telefone: '',
renda: 0.00, nascimento: new Date() };
      this.modoDeEdicao = true;
    }

    this.clienteForm = this.formBuilder.group({
      clienteid: [this.cliente.clienteid],
      nome: [this.cliente.nome, Validators.required],
      email: [this.cliente.email, Validators.required],
      telefone: [this.cliente.telefone, Validators.required],
      renda: [this.cliente.renda, Validators.required],
      nascimento: [this.cliente.nascimento, Validators.required],
      nascimentoForm: [{ value:
```

```
this.cliente.nascimento.toLocaleDateString(), disabled: !this.modoDeEdicao
}, Validators.required],
    });
  }
```

Precisamos garantir que nossa rota em nosso `app-routing.module.ts`, possua agora a previsão de um argumento, tal qual podemos ver na sequência, para que a página com os dados do cliente selecionado seja exibida.

```
{
    path: 'clientes-add-edit/:id',
    loadChildren: () => import('./pages/clientes/clientes-add-
edit/clientes-add-edit.module').then(m => m.ClientesAddEditPageModule)
},
```

Vamos testar nossa aplicação? Selecione um cliente na listagem e veja a página que já conhecemos para a inserção exibindo os dados do cliente que você selecionou.

Possibilitando a alteração dos dados

Como você verificou na execução anterior de nossa aplicação, notando a semelhança com as anteriores, nossos dados são exibidos, mas estão com seus controles desabilitados para alteração e temos, ao final da página, um botão que possibilitará alterar os dados destes clientes. Vamos então realizar as implementações para que este processo todo possa ocorrer. Observe que já implementamos o comportamento para o botão de alteração e para o botão de cancelamento da alteração. Já sabemos que estes métodos devem ser implementados na classe de renderização e nosso template, certo?

```
iniciarEdicao() {
    this.modoDeEdicao = true;
}

cancelarEdicao() {
    this.clienteForm.setValue(this.cliente);
```

```
    this.modoDeEdicao = false;
}
```

Para que possamos testar nossa aplicação, após a implementação anterior, precisamos inserir o código a seguir no componente de nosso botão de alteração que está no template HTML de nossa página. O segundo código é para o botão de cancelamento da edição, que também está em nosso template.

```
(click)="iniciarEdicao()"

(click)="cancelarEdicao()"
```

Antes de realizarmos o teste de atualização para os dados de cliente, precisamos realizar algumas modificações em nosso serviço e também no código controlador para o `submit()`.

Em relação ao nosso serviço, vamos criar um específico para alteração dos dados em `clientes.service.ts`, como apresentado na sequência. A atualização é simples, invocamos `setDoc()`, recebendo o que `doc()` retorna, que é o objeto da coleção `clientes`, que corresponde ao `clienteid`. Este objeto é então convertido e enviado para o `setDoc()` realizar a atualização. Importante saber que se o `clienteid` não existir, um novo documento é criado.

```
async update(cliente: Cliente) {
    await setDoc(doc(this._fireStore, "clientes", cliente.clienteid).
        withConverter(clienteConverter), cliente);
}
```

Em relação ao `submit()`, verifique na sequência o código a ser inserido, no local do código que já existir, após o `await loading.present();`, estamos substituindo o `then` pelo `await`.

```
if (this.cliente.clienteid === '') {
  await this.clientesService.create(this.clienteForm.value);
} else {
  await this.clientesService.update(this.clienteForm.value);
}
```

```
loading.dismiss().then(() => {
          this.toastService.presentToast('Gravação bem sucedida', 3000,
'top');
          this.router.navigateByUrl('/clientes-listagem');
});
```

Agora sim, vamos testar. Selecione um cliente na listagem, inicie a alteração e, em seguida, grave os novos dados.

A remoção de um cliente para fechamento

Temos nosso CRUD todo estruturado, restando-nos apenas implementar a remoção de um cliente da base de dados. Para isso, temos que implementar o serviço que realize essa operação, ajustar nossa listagem para oferecer ao usuário esta funcionalidade e implementá-la em nossa classe do template. Veja os códigos e neles os comentários.

```
// Método em clientes.service.ts.
async removeById(clienteId: string) {
    await deleteDoc(doc(this._fireStore, "clientes", clienteId));
}
```

No consumo deste serviço, adotaremos o uso de um alerta para solicitar a confirmação ao usuário para que a remoção possa efetivamente ser executada. Para isso, realizaremos uma implementação em nosso `AlertService`, onde criaremos o método apresentado na sequência.

```
async presentConfirm(header: string, message: string, successFunction)  {
    const alert = await this.alertCtrl.create({
        header,
        message,
        buttons: [
            {
                text: 'Cancel',
                role: 'cancel',
                handler: () => {
```

```
                    console.log('Remoção cancelada');
                }
            },
            {
                text: 'Manda brasa',
                handler: () => {
                    successFunction();
                }
            }
        ]
    });
    await alert.present();
}
```

Observou, na assinatura do método, que recebemos menos argumentos em relação ao método que já temos implementado? Para nosso objetivo, que é exibir uma mensagem ao usuário e realizar uma atividade com a confirmação em um dos botões que serão oferecidos, é o que basta. Mas notou o último atributo? Não o tipificamos, mas o nome é suscetível. Se notar mais adiante no código, verá que, caso o usuário confirme a remoção do cliente, essa variável é invocada como um método. Nosso método receberá uma função, mais precisamente, uma `Arrow Function`.

Vamos então implementar o método que consumirá nossos dois métodos recentemente implementados e que estará em nossa classe que renderizará a listagem. Veja-o na sequência. Note a criação da variável `successFunction`, recebendo uma `Arrow Function`, que é a funcionalidade que queremos que seja executada quando o usuário confirmar pela remoção do cliente. Neste código, estamos consumindo os dois serviços que implementamos, mas lembre-se de que o código de `successFunction` só será executado caso o usuário efetivamente confirmar a remoção. Injeções serão necessárias no construtor.

```
async removerCliente(cliente: Cliente) {
    try {
        const successFunction = async () => {
```

```
            this.clientesService.removeById(cliente.clienteid);
            this.toastService.presentToast('Cliente removido com sucesso',
3000, 'top');
            this.slidingList.closeSlidingItems();
            this.clientes = await this.clientesService.getAll();
        };
        await this.alertService.presentConfirm('Remover Cliente',
'Confirma remoção?', successFunction);
    } catch (e) {
        await this.alertService.presentAlert('Falha', 'Remoção não foi
executada', e, ['Ok']);
    }
}
```

Veja também, no código anterior, que utilizamos um `this.slidingList`, que ainda não temos. Vamos criá-lo antes do nosso construtor com o código a seguir. Importações serão necessárias. Observe a invocação ao `getAll()` após o sucesso na remoção. Isso atualizará a listagem.

```
@ViewChild('slidingList') slidingList: IonList;
```

Temos nosso CRUD completo! Vamos testar agora a remoção. Na listagem de clientes, busque a exibição do botão para remoção, deslizando o cliente desejado para a esquerda. Ao pressionar o botão, uma popup será exibida, solicitando a confirmação para a operação. Se a operação for confirmada, o cliente será removido e a listagem atualizada.

Atendimento com pesquisa ao cliente

Nós já registramos o atendimento para o cliente no capítulo anterior, fazendo uso do SQLite. Na seleção de clientes, utilizamos um controle, o `<ion-select>`, para exibir os clientes já registrados para escolhermos o do atendimento. Isso funcionou, mas comentamos que não é a melhor solução para selecionar um item quando temos muitas opções para compor a listagem. Veremos agora o uso de

uma técnica específica para pesquisa, mais comum em aplicações, sejam móveis, desktop ou web.

Na visão do atendimento, ao lado do controle que exibirá o nome do cliente, teremos um botão que, quando pressionado, abrirá uma nova visão com a listagem dos clientes recuperados e com um controle onde o usuário poderá digitar o início do nome do cliente, o que permitirá uma filtragem nos dados recuperados para que sejam exibidos apenas os que coincidam com o valor digitado. Esse controle é um `<ion-searchbar>` e veremos aqui o necessário para nossa implementação.

Entretanto, para implementarmos esta técnica, precisaremos adaptar o mecanismo de persistência para o atendimento, que passará a utilizar o Firestore, assim como fizemos para os clientes.

7.5 Visualização dos atendimentos registrados no Firestore

Seguindo nosso padrão de implementação, temos sempre a página com listagem dos dados e, a partir dela, inserimos, consultamos, alteramos e removemos. Vamos para a implementação da página responsável pela exibição de todos os atendimentos que já temos persistidos em nossa base de dados remota, no Firestore - de início, não temos nenhum. Trabalharemos em subseções, que apontarão cada arquivo que criaremos, ou adaptaremos para esta funcionalidade.

Método getAll() em OrdensDeServicoService

No capítulo anterior, implementamos o método `getAll()` em `OrdensDeServicoService`, que funcionou perfeitamente para o SQLite, agora precisamos adaptar para o Firestore, mas, antes disso, vamos ajustar nosso modelo para os conversores a serem utilizados no

Firestore. Veja o código a seguir a ser inserido em `ordemdeservico.model.ts`:

```
export const ordemDeServicoConverter = {
    toFirestore: (ordemDeServico) => {
        return <OrdemDeServico>{
            clienteid: ordemDeServico.clienteid,
            veiculo: ordemDeServico.veiculo,
            dataehoraentrada: ordemDeServico.dataehoraentrada,
        };
    },
    fromFirestore: (snapshot, options) => {
        const data = snapshot.data(options);
        return <OrdemDeServico>{
            ordemdeservicoid: snapshot.id,
            clienteid: data.clienteid,
            veiculo: data.veiculo,
            dataehoraentrada: data.dataehoraentrada.toDate(),
        }
    }
};
```

Agora, na sequência, veja a nova implementação para o método. Lembre-se de que ele deverá estar em `ordensdeservico.service.ts`. Antes do método, temos o construtor recebendo a injeção do Firestore.

```
// imports
import { Injectable } from '@angular/core';
import { collection, Firestore, getDocs, orderBy, query } from
'@angular/fire/firestore';
import { OrdemDeServico, ordemDeServicoConverter } from
'../models/ordemdeservico.model';
s
// Construtor
constructor(
        private _fireStore: Firestore,
    ) { }
```

```
// getAll

public async getAll(): Promise<OrdemDeServico[]> {
    const ordensDeServico: OrdemDeServico[] = [];
    const q = query(collection(this._fireStore, "ordensdeservico"),
orderBy("dataehoraentrada",
"desc")).withConverter(ordemDeServicoConverter);
    const querySnapshot = await getDocs(q);
    querySnapshot.forEach((doc) => {
        ordensDeServico.push(doc.data());
    });
    return ordensDeServico;
}
```

Veja que tal qual fizemos para clientes, recuperamos os dados da coleção e os adicionamos a uma matriz, que então é retornada ao chamador. Ainda precisamos retirar/comentar os demais métodos do serviço, pois mudamos a estratégia de persistência.

A classe que fornecerá interação para o template de listagem

Precisaremos adaptar o código da classe `OrdensDeServicoListagemPage` do arquivo `ordensdeservico-listagem.page.ts` para que utilize nossa implementação do código anterior, que fez uso do Firestore para recuperação de dados. Tecnicamente é semelhante ao que implementamos anteriormente para clientes, mas vamos verificar na sequência o código específico para esta seção, com alguns comentários entre eles. A explanação sobre as implementações é a mesma que vimos quando implementamos essa funcionalidade para clientes.

```
// Redefinição da propriedade para o tipo específico para o Firestore
public ordensDeServico: OrdemDeServico[] = [];


// Adaptação do método para recuperar os dados do novo serviço
async ionViewWillEnter() {
```

```
    this.ordensDeServico = await this.ordensDeServicoService.getAll();
}
```

Vamos testar nossa aplicação. Selecione `Atendimentos` no menu e veja que a listagem de atendimentos aparece vazia, o que é natural. Não temos nada registrado ainda no Firestore.

7.6 A pesquisa de clientes para o atendimento

Para que possamos implementar a funcionalidade comentada anteriormente para a realização de pesquisa de clientes para o registro de um atendimento, precisamos implementar todo um componente que oferecerá a interação com o usuário. Seguindo nossa estratégia para explanação, vamos trabalhar com subseções, e cada uma apresentará um artefato para este componente que criaremos.

O template para a página de pesquisa

No terminal, na pasta de nosso projeto, vamos executar a instrução a seguir para que todos os artefatos que precisamos sejam criados e então, após isso, trabalharemos nos ajustes de cada um deles.

```
ionic generate page pages/clientes/clientes-search
```

Com os arquivos criados, vamos ajustar o `clientes-search.page.html` tal qual o código a seguir.

```
<ion-header>
    <ion-toolbar>
      <ion-buttons slot="start">
      </ion-buttons>
      <ion-title>Pesquisar por CLIENTE</ion-title>
    </ion-toolbar>
</ion-header>
```

```
<ion-content no-padding>
    <ion-searchbar placeholder="Nome para filtrar"
(ionInput)="getClientes($event)"></ion-searchbar>
    <ion-list *ngFor="let cliente of clientes"
(click)="clientesClick(cliente)">
        <ion-item>
            <ion-label>
                {{cliente.nome}}
            </ion-label>
        </ion-item>
    </ion-list>
</ion-content>
```

O que temos de novo no código anterior? Conseguiu identificar? Exatamente, o uso de `<ion-searchbar>`. É este componente que permitirá a renderização dos controles relacionados à pesquisa. Temos várias propriedades e eventos que podemos configurar para este componente, mas estamos utilizando apenas o `placeholder`, para um texto ser exibido ao usuário, antes que ele comece a digitação, e o `ionInput`, que será disparado a cada tecla pressionada para compor o nome a ser pesquisado.

Abaixo do `<ion-searchbar>`, temos o já conhecido `<ion-list>`, responsável pela exibição dos clientes já cadastrados, que será filtrada de acordo com o que for sendo informado para pesquisa pelo usuário. Mas como esta filtragem ocorre?

Em primeiro lugar, não há ligação direta entre o controle de pesquisa e a listagem. Não há mágica, precisamos programar. Entretanto, podemos identificar que a listagem, como nos exemplos anteriores, é composta pela varredura em uma variável, chamada `clientes` em nosso caso. Também conseguimos identificar um método chamado `getClientes()`, ligado ao evento `ionInput()`, que receberá um `$event`, que, por sua vez, levará informações do controle `<ion-searchbar>` para nosso código. É neste método que implementaremos o processo de captura do que for digitado pelo usuário, para então podermos realizar a seleção dos clientes.

Um novo serviço em ClientesService

Antes de implementarmos a classe que será responsável pela renderização do template anterior, precisamos codificar o método para seleção dos clientes que tenham o nome iniciado pelos valores que serão digitados pelo usuário no controle de pesquisa. Para isso, vamos trabalhar na classe `ClientesService`, que está no arquivo `clientes.service.ts`, na pasta `src/app/services`.

O seguinte código deverá ser implementado ao final da classe. Observe que o método recebe uma string, que conterá o nome, ou o início dele, que deverá ser utilizada como critério de seleção. O retorno do método é o mesmo que definimos para o `getAll()`, que, inclusive, invocamos na primeira instrução do método, populando nossa coleção a ser trabalhada. Caso nenhum valor chegue no argumento `nome`, retornamos a coleção inteira.

```
async getByNome(nome: string): Promise<Cliente[]> {
    const clientes: Cliente[] = await this.getAll();

    if (!nome) {
        return clientes;
    }

    return clientes.filter(
        cliente =>
cliente.nome.toLowerCase().startsWith(nome.toLowerCase()));
}
```

E se o argumento `nome` trouxer valores? Aí sim precisamos processá-lo em nossa coleção. Aplicamos a seleção pela invocação ao método `filter()`, utilizando um filtro com base no método `startsWith()` dos caracteres em minúsculos.

A classe TypeScript para nosso template

Vamos agora para a adaptação de nossa classe responsável pela renderização de nosso template HTML. Veja o código dela logo na

sequência. É o arquivo com o nome `clientes-search.page.ts` .

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { Cliente } from 'src/app/models/cliente.model';
import { ClientesService } from 'src/app/services/clientes.service';
import { Location } from '@angular/common';

@Component({
  templateUrl: './clientes-search.page.html',
  styleUrls: ['./clientes-search.page.scss']
})
export class ClientesSearchPage implements OnInit {
  public clientes: Cliente[];

  constructor(
    private clientesService: ClientesService,
    public location: Location
  ) { }

  ngOnInit() {
  }

  async ionViewWillEnter() {
    this.clientes = await this.clientesService.getAll();
  }

  async getClientes(searchBar) {
    this.clientes = await
this.clientesService.getByNome(searchBar.srcElement.value);
  }

  clientesClick(cliente: Cliente) {
    this.location.back();
  }
}
```

No código anterior, observe que no decorator `@Component()` estamos definindo um arquivo para `styleUrls` , onde definiremos um estilo

específico para nossa listagem de clientes. Mais à frente veremos este arquivo.

Dentro da classe, temos a definição da propriedade `clientes`, que não é algo novo para nós, pois estamos sempre trabalhando com este tipo de propriedade quando nos referimos a listagens.

No construtor, temos a injeção de `ClientesService`, que possui o serviço de seleção que implementamos recentemente, além do `getAll()`, que invocamos no momento em que a visão recebe o foco, para que todos os clientes possam ser exibidos já na primeira exibição. Isso ocorre no método `ionViewWillEnter()`. Você pode pensar em não fazer uso desta técnica para economizar tráfego desnecessário, pois pode não ser seu objetivo exibir os clientes sem um valor informado para o filtro.

Na sequência, implementamos o método `getClientes()`, recebendo nosso `SearchBar`, onde invocamos o método `getByNome()`, enviando a ele o valor que está digitado no controle. Este método retornará os clientes já com a seleção baseada no valor informado pelo usuário.

O último método, `clientesClick()`, que será invocado quando o usuário escolher um cliente da listagem, a princípio apenas faz uso do objeto `location`, também injetado em nosso construtor, para que a aplicação retorne para a visão anterior à de pesquisa. Logo veremos tudo isso em funcionamento.

O arquivo com definição de estilos para nosso template

Chegamos à etapa final para a implementação de nosso componente de pesquisa, onde vamos atualizar o arquivo `clientes-search.page.scss` inserindo nele o pequeno código que vemos na sequência. Já conversamos anteriormente sobre este tipo de arquivo e sobre CSS, mas para informação rápida, esta configuração deixará os itens da listagem menores em relação à altura. Quando formos testar nossa aplicação, podemos fazê-lo com este arquivo, para identificarmos a diferença.

```
ion-list {
    margin-bottom: 0% !important;
    margin-top: 0% !important;
}
```

7.7 A criação de um atendimento com acesso à pesquisa por clientes

Ainda não podemos testar nossa aplicação, pois o acesso à visão de pesquisa que criamos anteriormente estará no momento de registro do atendimento, seja na criação de um novo ou alteração de um já existente. Nós temos esta funcionalidade já implementada e a testamos no capítulo anterior, mas o fizemos com outra técnica para seleção do cliente e, ainda, utilizamos o SQLite. Agora, estamos mudando o mecanismo de persistência e o controle para seleção de clientes. Vamos ver estas alterações nas subseções seguintes.

O template para a página do CRUD para atendimento

Como estamos adaptando um código já trabalhado, não trarei para cá todo o template para o CRUD de atendimentos, mas apenas o conteúdo que precisa ser alterado, que, especificamente é a parte onde o cliente deve ser informado. Veja o código a seguir, que está no arquivo `ordensdeservico-add-edit.page.html` .

```
<ion-item lines="none" [disabled]="!modoDeEdicao">
    <ion-label position="floating" class="label-input-title"
style="margin-top: 10%;">Cliente</ion-label>
    <ion-grid no-padding>
        <ion-row no-padding>
            <ion-col size="11" no-padding disabled="true">
                <ion-input value={{nomeCliente}} type="text"
```

```
style="margin-left: -8%;"></ion-input>
            </ion-col>
            <ion-col size="1" no-padding>
                <ion-button routerDirection="forward" [routerLink]="['/',
'clientes-search']">
                    <ion-icon slot="icon-only" name="search"size="small">
</ion-icon>
                </ion-button>
            </ion-col>
        </ion-row>
    </ion-grid>
</ion-item>
```

Veja que nosso primeiro `<ion-col>` está desabilitado. Observe, no segundo `<ion-col>`, que temos um botão com uma rota que será configurada mais à frente para nossa página de pesquisa, que criamos agora há pouco. Veja também que no `<ion-input>` estamos realizando a ligação por meio das `{{}}`, que já utilizamos antes.

Para o `<ion-button>`, estamos invocando um método, que nos redirecionará para a página de pesquisa, tal qual estamos utilizando em nossas navegações.

Algumas mudanças precisarão ser realizadas em nosso projeto. Vamos a elas.

1. Retire o `import { Guid } from 'guid-typescript';` dos imports no arquivo `ordensdeservico-add-edit.page.ts`;
2. Comente, no mesmo arquivo, todo o conteúdo dos métodos `ionViewWillEnter()` e `submit()`;
3. Deixe seu método `ngOnInit()` como o apresentado na sequência:

```
async ngOnInit() {
 this.ordemDeServico = {ordemdeservicoid: '',
     clienteid: '', veiculo: '', dataehoraentrada: new Date() };
```

```
 this.osForm = this.formBuilder.group({
     ordemdeservicoid: [this.ordemDeServico.ordemdeservicoid],
     clienteid: [this.ordemDeServico.clienteid, Validators.required],
     veiculo: [this.ordemDeServico.veiculo, Validators.required],
     dataentrada: [{ value:
this.ordemDeServico.dataehoraentrada.toLocaleDateString(), disabled:
!this.modoDeEdicao }, Validators.required],
     horaentrada: [{ value:
this.ordemDeServico.dataehoraentrada.toLocaleTimeString(), disabled:
!this.modoDeEdicao }, Validators.required],
     dataehoraentrada: [this.ordemDeServico.dataehoraentrada]
 });

 this.modoDeEdicao = true;
}
```

4. No arquivo `ordensdeservico-add-edit.page.ts`, insira antes do construtor a declaração `public nomeCliente= 'Pesquise o cliente';`.

Antes de executarmos a aplicação, precisamos registrar em `app.module.ts`, em `providers`, o `DatePipe` e realizar a importação necessária para ele.

Agora sim, execute e veja na execução da sua aplicação se a visão do atendimento está semelhante à figura a seguir. Lembre-se de que será preciso pressionar `Atendimentos` no menu e depois o botão de inserir novo atendimento para chegar na página da figura.

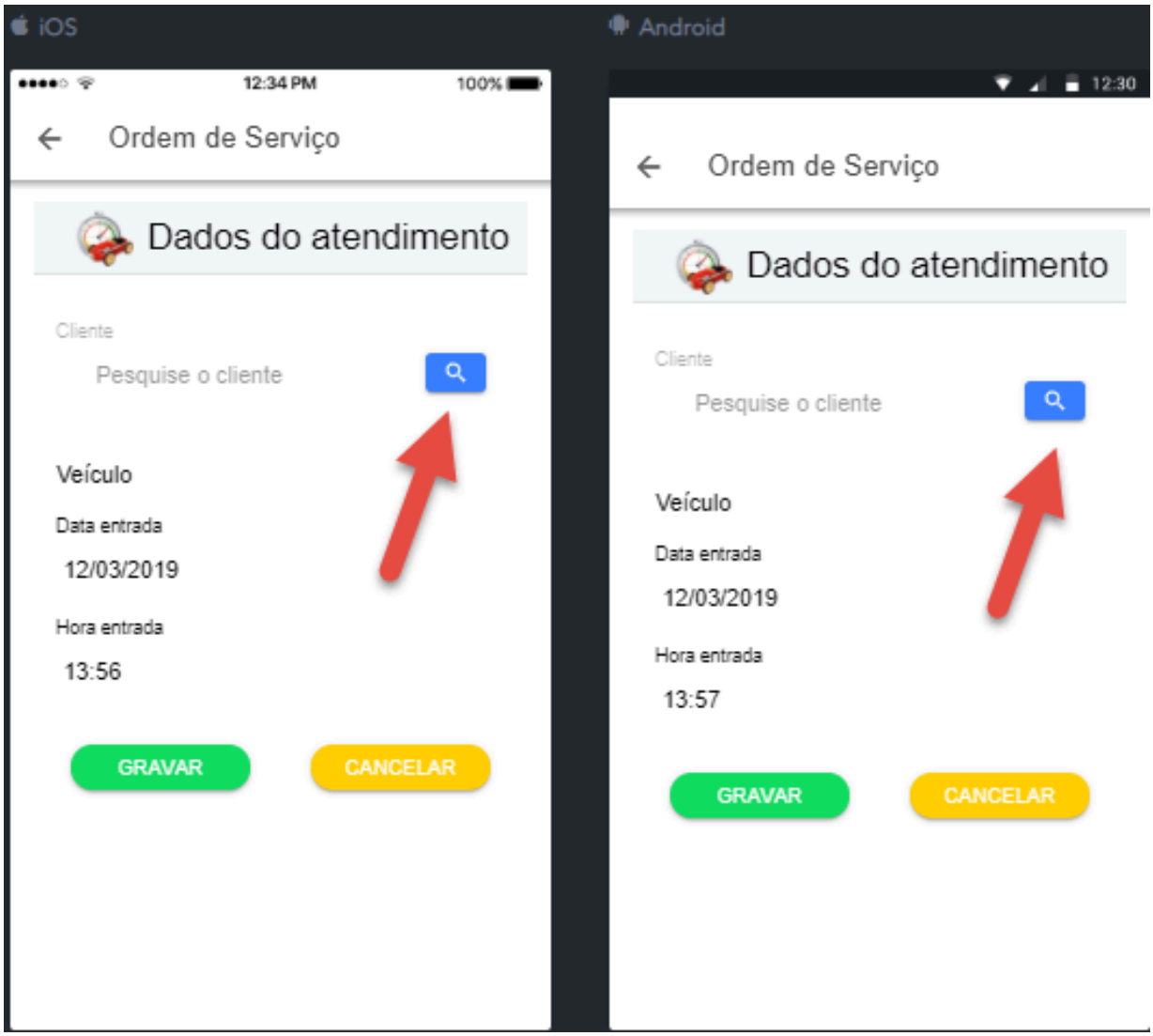


Figura 7.3: Página com a interface para inserção de atendimentos

Viu o destaque na figura anterior para o botão que dispara a pesquisa de clientes? Clicou nele? Se sim, veja na figura a seguir como deve estar sua aplicação.

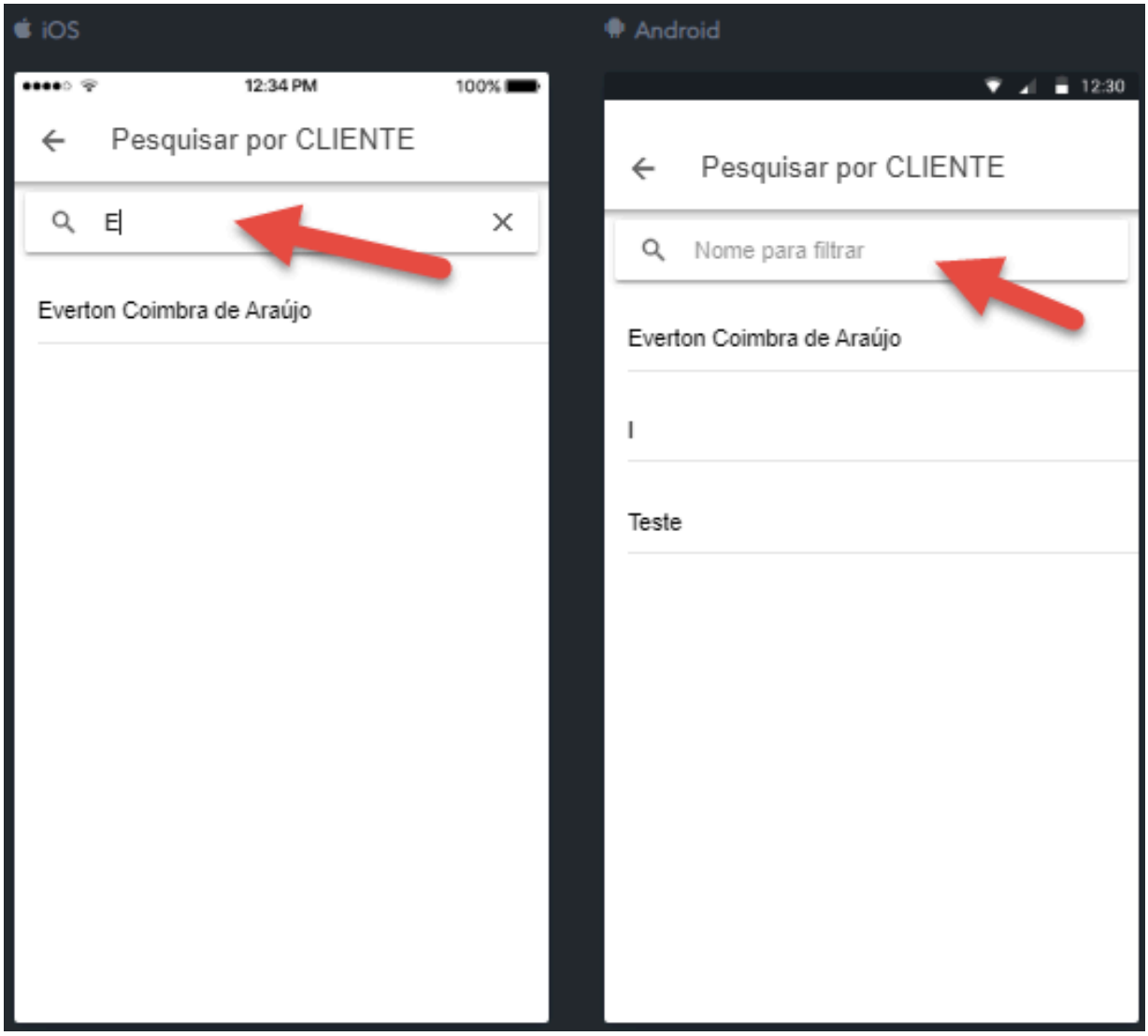


Figura 7.4: Página com a pesquisa de clientes

Notou que no Android não existe valor digitado na pesquisa, o que leva a exibição de todos os clientes registrados? E já no iOS a listagem está exibindo apenas os nomes que começam com E ?

Mas a ideia seria pressionarmos no nome do cliente desejado e que o nome dele fosse exibido em nossa visão de registro de atendimento e isso ainda não está funcionando. Mas vamos implementar esta funcionalidade já na sequência, na próxima seção.

7.8 Eventos na comunicação entre componentes de visão

Em alguns momentos de nossa aplicação teremos a necessidade de comunicação entre uma `tela` e outra (usei o termo `tela` agora para referenciar a visão, nosso template). No caso específico que estamos implementando, no momento em que estivermos pesquisando por um cliente, ao selecionarmos o desejado, precisamos passar para o componente de registro de atendimento este cliente. Vamos fazer isso? É bem simples.

Em primeiro lugar, precisamos criar um serviço para trabalhar com as pesquisas. Para isso, crie em `services` um arquivo chamado `search-service.ts` e nele implemente o código a seguir:

```
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';

@Injectable({
    providedIn: 'root'
})
export class SearchService {
    private searchSubject = new Subject<any>();

    publishSomeData(data: any) {
        this.searchSubject.next(data);
    }

    getObservable(): Subject<any> {
        return this.searchSubject;
    }
}
```

Observe que estamos fazendo uso do `RxJs` para manipularmos a comunicação por meio de `Subject` e `Observable`. Temos, no código anterior, uma propriedade privada chamada `searchSubject` para qualquer tipo de dado, o que a torna genérica. Depois, temos dois métodos, o primeiro, que publicará em nosso `Subject` um dado

recebido. O segundo método, o `getObservable()`, retornará o `Subject`, com o dado armazenado nele.

Agora precisamos consumir essa classe e começaremos a fazer isso injetando o novo serviço no construtor de nossa classe `OrdensDeServicoAddEditPage`, que está no arquivo `ordensdeservico-add-edit.page.ts`, tal qual o código a seguir.

```
private searchService: SearchService,
```

Agora precisamos implementar um método para nos registrarmos no serviço injetado e faremos isso por meio do código a seguir, que deve ser implementado na mesma classe.

```
private registrarServicoClienteSelecionado() {
    this.searchService.getObservable().subscribe((data: Cliente) => {
      this.nomeCliente = data.nome;
      this.osForm.controls.clienteid.setValue(data.clienteid);
    });
}
```

Observe a invocação do método `subscribe()` após a invocação ao `getObservable()` do serviço `SearchService`, representado pela variável `this.searchService`. Verifique que estamos tipificando `data` com `Cliente` e que, quando houver um retorno, atribuímos à variáveis locais a classe, propriedades do objeto recebido. Após a implementação, precisaremos invocar esse novo método em nosso construtor, inserindo nele o código a seguir.

```
this.registrarServicoClienteSelecionado();
```

Agora precisamos implementar o envio do objeto para o `Observable` registrado anteriormente. Faremos isso no momento em que um cliente for selecionado em nossa pesquisa, no `clientes-search.page.ts`. Veja o método a seguir, que já existe, com esta funcionalidade adicionada, com comentários na sequência do código. Precisaremos realizar a injeção para o serviço.

```
clientesClick(cliente: Cliente) {
    this.searchService.publishSomeData(cliente);
    this.location.back();
}
```

Podemos testar novamente nossa aplicação. Agora, na visão de pesquisa do cliente, pressione o encontrado e verifique o retorno para a página de registro do atendimento e que, agora, o nome do cliente selecionado aparece no controle.

É preciso uma implementação para finalizarmos esta etapa referente à pesquisa. Nós registramos a assinatura ao evento, mas precisamos agora cancelar esta assinatura, caso contrário, a aplicação estará sempre escutando por algo e, ao sairmos da página de registro de atendimento, não terá mais essa necessidade. Para isso, veja a implementação a seguir, que deve ser implementada em `ordensdeservico-add-edit.page.ts`.

```
public unsubscribeServices() {
    this.searchService.getObservable().unsubscribe();
}
```

Verifique que estamos implementando um método, que poderá ser o responsável por cancelar a assinatura aos eventos que tenhamos registrados. Mas quando chamaremos este método? Poderíamos pensar em algum evento do ciclo de vida da visão, mas isso nos daria um certo trabalho no momento, pois precisamos controlar quando registrar e cancelar, já que fazemos uso dele em uma visão chamada por essa assinatura.

Existe o método `ngOnDestroy()`, mas, pela estratégia que estamos utilizando para navegação, esse método não será chamado. Vamos então invocá-lo quando o botão de retornar para a visão anterior for pressionado. Veja a nova implementação para o controle na sequência. Observe a ligação com o evento `click`, em `ordensdeservico-add-edit.page.html`.

```
<ion-back-button defaultHref="ordensdeservico-listagem"
(click)="unsubscribeServices()"></ion-back-button>
```

Temos aqui um problema: caso o usuário acesse a inserção/edição, retorne para a listagem e a acesse novamente. Com o `unsubscribe()`, o objeto retornado pelo `getObservable()` estará fechado. Precisamos ajustar o método para trabalhar essa situação. Veja o ajuste dele, no arquivo `search.service.ts` na sequência.

```
getObservable(): Subject<any> {
    if (this.searchSubject.closed) {
        this.searchSubject = new Subject<any>();
    }
    return this.searchSubject;
}
```

7.9 O registro do atendimento com o cliente selecionado

Já temos os dados para o atendimento podendo ser devidamente informados pelo usuário. Precisamos agora realizar a gravação deles em nossa base, no Firestore. Já fizemos um processo semelhante quando trabalhamos com clientes, anteriormente no início deste capítulo. Vamos então a algumas adaptações necessárias para que o processo possa ser realizado agora para atendimentos.

Método update() em OrdensDeServicoService

Toda contextualização sobre a atualização no Firestore já foi realizada anteriormente, assim como a explanação das mudanças necessárias do SQLite para o Firestore. Desta maneira, vamos direto ao novo código para o método `update()`, da classe `OrdensDeServicoService`, que está no arquivo `ordensdeservico.service`. Veja-o na sequência e depois comentarei sobre ele. Você precisará realizar imports para o `setDoc()` e `doc()`.

```
async update(ordemDeServico: OrdemDeServico) {
    ordemDeServico.dataehoraentrada = new
Date(ordemDeServico.dataehoraentrada);
    const clientesRef = collection(this._fireStore, "ordensdeservico");

    if (ordemDeServico.ordemdeservicoid.length == 0) {
        await setDoc(doc(clientesRef).
            withConverter(ordemDeServicoConverter), ordemDeServico);

    } else {
        await setDoc(doc(this._fireStore, "ordensdeservico",
ordemDeServico.ordemdeservicoid).
            withConverter(ordemDeServicoConverter), ordemDeServico);
    }
}
```

Veja que estamos persistindo o objeto recebido, verificando se devemos atualizar ou criar um novo, tendo como base a propriedade `ordemdeservicoid`.

A invocação do update() no submit() do template

Já temos o serviço que realiza a gravação, por enquanto, trabalhando a inserção no Firestore. Precisamos agora invocá-lo. Já sabemos, por práticas anteriores, que esse processo é realizado por meio do `submit` do formulário, onde ligamos um evento também chamado, por convenção, de `submit()`. Vamos então implementá-lo, tal qual pode ser verificado no código a seguir.

```
async submit() {
    if (this.osForm.invalid || this.osForm.pending) {
      await this.alertService.presentAlert('Falha', 'Gravação não foi
executada',
        'Verifique os dados informados para o atendimento', ['Ok']);
      return;
    }

    const loading = await this.loadingCtrl.create();
    await loading.present();
```

```
    const data = new
Date(this.osForm.controls.dataentrada.value).toISOString();
    const hora = new
Date(this.osForm.controls.horaentrada.value).toISOString();

    this.osForm.controls.dataehoraentrada.setValue(
      data.substring(0, 11) + hora.substring(11, 22)
    );

    await this.ordensDeServicoService.update(this.osForm.value);

    loading.dismiss().then(() => {
      this.toastService.presentToast('Gravação bem sucedida', 3000,
'top');
      this.router.navigateByUrl('ordensdeservico-listagem');
    });

}
```

Verifique, antes da invocação ao `update()`, uma instrução que invoca o método `setValue()` para `dataehoraentrada`. Precisamos mapear a data e hora que temos no `FormGroup` para uma data completa, pois é ela que armazenaremos. Antes disso, criamos duas variáveis, `data` e `hora`, que nos auxiliarão neste processo. O trabalho com datas em JavaScript, em consequência com o TypeScript, não é algo muito trivial. Veja que recuperamos o valor de cada controle e o concatenamos antes de atribuir. Atente para a injeção necessária para o `firestoreService` e `loadingCtrl`.

7.10 A alteração de um atendimento já registrado

Relembrando, nós já temos praticamente toda a implementação necessária para ordens de serviço, pois a trabalhamos no capítulo

anterior. O que precisamos é adaptar estas implementações para o uso do Firestore. Faremos isso começando pela refatoração do método `ionViewWillEnter()`, onde retiraremos dele a responsabilidade de criação do FormGroup, para um método específico. Veja este método na sequência, lembrando que nosso arquivo é o `ordensdeservico-add-edit.page.ts`.

```
private createUpdateFormGroup() {
    this.osForm = this.formBuilder.group({
        ordemdeservicoid: [this.ordemDeServico.ordemdeservicoid],
        clienteid: [this.ordemDeServico.clienteid, Validators.required],
        veiculo: [this.ordemDeServico.veiculo, Validators.required],
        dataentrada: [this.ordemDeServico.dataehoraentrada.toISOString(),
Validators.required],
        horaentrada:
[this.ordemDeServico.dataehoraentrada.toLocaleTimeString('pt-BR'),
Validators.required],
        dataehoraentrada: ['']
    });
}
```

Com a alteração que realizamos para a atribuição da hora, que trouxe a possibilidade de termos essa atribuição localizada, precisamos ajustar o `submit()` na parte de atribuição do valor para a `dataehoraentrada` para o modelo de negócio. Veja na sequência. Com ela, não precisamos mais da variável `hora`.

```
this.osForm.controls.dataehoraentrada.setValue(
  data.substring(0, 11) + this.osForm.controls.horaentrada.value
);
```

Agora, levaremos para o método `ngOnInit()` a invocação ao método criado anteriormente. Mas vamos trazer aqui todo o método para relembrar. Veja-o na sequência, e, após a listagem, trago algumas explicações.

```
async ngOnInit() {
    const id = this.route.snapshot.paramMap.get('id');
```

```
    if (id !== '-1') {
        this.ordemDeServico = await
this.ordensDeServicoService.getById(id);
        const cliente = await
this.clientesService.getById(this.ordemDeServico.clienteid);
        this.nomeCliente = cliente.nome;
    } else {
        this.ordemDeServico = {ordemdeservicoid: '', clienteid: '',
veiculo: '', dataehoraentrada: new Date() };
        this.modoDeEdicao = true;
    }

    this.createUpdateFormGroup();
}
```

Por que separamos os métodos? Primeiro, porque para uma boa coesão isso é importante, a separação de responsabilidades. A segunda explicação é que, quando formos cancelar uma operação de alteração, precisaremos atualizar o `FormGroup` para os dados originais do atendimento e, no objeto de atendimento, a variável `this.ordemDeServico` não tem os campos `dataentrada` e `horaentrada`, que compomos com base no `dataehoraentrada`, como pode ser visto nos códigos anteriores. Observe também, no último código, que invocamos `getById()` de `clientesService`, que precisaremos injetar no construtor. Fazemos isso por precisar do nome do cliente, para que seja exibido na visão.

Para completarmos a implementação da alteração, precisamos adaptar o método que será disparado quando o usuário clicar no botão de cancelamento da alteração. Veja este método na sequência.

```
cancelarEdicao() {
    this.createUpdateFormGroup();
    this.modoDeEdicao = false;
}
```

Nós já temos em `ordensdeservico.service.ts` o método `getById()` implementado, mas para uso do SQLite. Agora precisamos adaptar

para o Firestore. Veja-o então no código a seguir. Praticamente é o mesmo comportamento que implementamos para clientes. Temos como diferencial o trabalho com a data recuperada, mas é algo facilmente compreendido. Será preciso importar o `getDoc().

```
async getById(clienteId: string): Promise<OrdemDeServico> {
    const q = doc(this._fireStore, "ordensdeservico",
clienteId).withConverter(ordemDeServicoConverter);
    const querySnapshot = await getDoc(q);

    return querySnapshot.data();
}
```

7.11 A remoção de um atendimento já registrado

Esta etapa é a final para nosso trabalho deste capítulo, e é bem simples, pois só precisamos mudar nosso serviço, em `ordensdeservico.service.ts`, tal qual podemos ver na sequência. Será preciso importar o `deleteDoc()`.

```
async removeById(ordensDeServicoId: string) {
    await deleteDoc(doc(this._fireStore, "ordensdeservico",
ordensDeServicoId));
}
```

Também precisamos ajustar nosso método `removerAtendimento()` em `ordensdeservico-listagem.page.ts

```
async removerAtendimento(ordemDeServico: OrdemDeServico) {
    try {
      const successFunction = async () => {
        await
this.ordensDeServicoService.removeById(ordemDeServico.ordemdeservicoid);
        this.toastService.presentToast('Atendimento removido com sucesso',
3000, 'top');
```

```
        this.slidingList.closeSlidingItems();
        this.ordensDeServico = await this.ordensDeServicoService.getAll();
      };
      await this.alertService.presentConfirm('Remover Atendimento',
'Confirma remoção?', successFunction);
    } catch (e) {
      await this.alertService.presentAlert('Falha', 'Remoção não foi
executada', e, ['Ok']);
    }
}
```

Agora podemos executar e testar nossa aplicação criando um novo atendimento, caso nenhum exista. Depois, na listagem, selecione um atendimento já registrado, ele será exibido como consulta, pode alterá-lo e então tentar gravar e cancelar. Depois, seguindo o procedimento de seleção de um item na listagem de atendimentos, remova um.

Se você quiser testá-la nos dispositivos ou emuladores, basta utilizar as instruções que já trabalhamos em capítulos anteriores.

Como registro, a aplicação implementada neste capítulo foi testada em emuladores e dispositivos Android e iOS, em ambientes de desenvolvimento Windows e Mac.Você pode também retirar o código de conexão com SQLite que temos no `app.component.ts` .

Conclusão

Terminamos mais este capítulo. Adaptamos nossa aplicação do capítulo anterior, que fazia persistência local no dispositivo para uma persistência distribuída, com o Firestore, mantendo nossos dados na nuvem e exigindo que nosso dispositivo esteja sempre conectado à internet. Podemos unir as duas tecnologias e pensarmos em uma sincronização para trabalho off-line, mas fica o desafio para você essa implementação.

Na nova implementação do registro de atendimentos (ordens de serviços), na seleção do cliente que será atendido, utilizamos uma

nova técnica, a de termos uma visão específica para pesquisa por um cliente, com base em valores informados. Não mais apenas selecionamos um cliente de um componente de listagem, como fizemos no capítulo anterior.

Aprendemos também a utilizarmos eventos para promovermos comunicação entre visões, com envios de objetos. Fizemos isso na seleção de um cliente na visão de pesquisa para registro dele no atendimento.

No próximo capítulo trabalharemos com o uso da câmera e álbum de fotografias.

Capítulo 8
Câmera e álbum de fotos no CRUD de clientes e Storage do Firebase

Começamos este novo capítulo com uma implementação de persistência na nuvem realizada. Fizemos isso no capítulo anterior, onde implementamos um CRUD no Firestore, produto do Firebase, do Google. Comentamos sobre a importância de termos uma conexão com a internet para esses serviços, o que é cada vez mais comum.

Em capítulos anteriores realizamos a implementação de CRUD com persistência local também, pois a conexão pode não ser garantida e nosso cliente pode precisar utilizá-la, com vistas a sincronizar os dados locais com a nuvem e também da nuvem com o dispositivo.

Trabalhamos apenas dados textuais, mas sabemos que a existência de imagens para auxiliar na experiência do usuário ou representar pessoas, objetos ou situações é cada vez mais constante. Sabendo disso, não podemos deixar de oferecer funcionalidades para interação da aplicação com a câmera e álbum de fotos dos dispositivos. Veremos isso neste capítulo, que você poderá constatar que é bem interessante.

Iniciaremos o processo interagindo com a câmera, trabalhando com a foto tirada pelo usuário, armazenando-a no dispositivo, de maneira física no ambiente destinado à aplicação, depois, publicaremos a imagem em um `Storage` do `Firebase`, tecnologia e recurso cada vez mais presente em `Cloud`.

Na implementação deste capítulo estou seguindo a que já temos concluída até o capítulo anterior.

8.1 Adaptação de nossa página de clientes

Trabalharemos aqui no livro a implementação com fotos na nossa página de clientes. Permitiremos que o usuário tire uma foto pela câmera, ou selecione uma já armazenada na galeria de fotos do dispositivo.

Para possibilitarmos esta funcionalidade ao nosso usuário, precisamos adaptar nossa interface com ele para a página de CRUD de Clientes, onde teremos uma área para a exibição da imagem e para os botões de interação para ativação da câmera ou seleção no álbum.

Antes de começarmos essa interação, vamos refletir sobre nossa página de clientes. As figuras que vimos no livro demonstram emuladores com telas pequenas e o que já temos como dados para o cliente praticamente ocupa toda a tela. Veremos uma adaptação do `<ion-input>` para ganharmos espaço e podermos ter estes novos controles disponibilizados em nossa página. Também retiraremos a imagem que tínhamos no topo da página. Desta maneira, veja a figura a seguir, com a alteração que implementaremos.

iOS
12:34 PM
100%
Back
Clientes
Nome
Email
Telefone
Renda
0
Nascimento
05/04/2019
Capturar Foto
Gravar
Cancelar

Figura 8.1: Página do CRUD de clientes com controles para a foto do cliente

Nos capítulos anteriores já tínhamos implementado a página com os controles que interagem com o usuário para solicitar os dados de cada cliente, que precisam ser informados para seu registro na aplicação. Agora, vamos redesenhar nossa interface com o usuário. Veja o código inicial para a nova implementação no código a seguir, que é semelhante ao que já tínhamos. Lembre-se de que este código está no arquivo `clientes-add-edit.page.html`, na pasta `src/app/pages/clientes/clientes-add-edit`.

```
<ion-header>
    <ion-toolbar>
        <ion-buttons slot="start">
            <ion-back-button defaultHref="/menu/(menucontent:clientes)">
</ion-back-button>
        </ion-buttons>
        <ion-title>Clientes</ion-title>
    </ion-toolbar>
</ion-header>

<ion-content padding>
    <ng-container *ngIf="clientesForm">
        <form [formGroup]="clientesForm" (submit)="submit()">
            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-input formControlName="nome" type="text"
placeholder="Nome"></ion-input>
            </ion-item>

            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-input formControlName="email" type="email"
placeholder="Email"></ion-input>
            </ion-item>

            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-input formControlName="telefone" type="tel"
placeholder="Telefone"></ion-input>
            </ion-item>
```

```
            <ion-item lines="none" [disabled]="!modoDeEdicao">
                <ion-label position="fixed" class="label-input-
title">Renda</ion-label>
                <ion-input formControlName="renda" type="number"></ion-
input>
            </ion-item>

            <div (click)="selecionarData()">
                <ion-grid class="ion-no-padding">
                  <ion-row>
                    <ion-col size="9">
                      <ion-item lines="none">
                        <ion-label position="stacked">Nascimento</ion-
label>
                        <ion-input formControlName="nascimentoForm"></ion-
input>
                      </ion-item>
                    </ion-col>
                    <ion-col class="ion-no-padding">
                      <ion-item lines="none">
                        <ion-button fill="clear" class="ion-no-margin;ion-
no-padding">
                          <ion-icon icon="calendar"></ion-icon>
                        </ion-button>
                      </ion-item>
                    </ion-col>
                  </ion-row>
                </ion-grid>
            </div>

            <ion-button *ngIf="!modoDeEdicao" shape="round"
color="primary" expand="block" padding (click)="iniciarEdicao()">
                Alterar dados
            </ion-button>
            <div *ngIf="modoDeEdicao">
                <ion-grid>
                    <ion-row>
                        <ion-col col-6 >
                            <ion-button class="nopadding" shape="round"
```

```
color="success" size="small" expand="block" type="submit">
                                Gravar
                            </ion-button>
                        </ion-col>

                        <ion-col col-6>
                            <ion-button class="nopadding"  shape="round"
color="warning" size="small" expand="block" (click)="cancelarEdicao()">
                                Cancelar
                            </ion-button>
                        </ion-col>
                    </ion-row>
                </ion-grid>
            </div>
        </form>
    </ng-container>
</ion-content>
```

Notou o início do template, em que tiramos a imagem de cabeçalho? Conseguiu verificar também que não temos mais todos os `<ion-label>` ? E nos que ficaram, notou que mudamos `position` para `fixed` em `renda` ? Conseguiu identificar que definimos uma classe de estilo para os `<ion-button>` de `Gravar` e `Cancelar` , chamada `nopadding` ? Tudo para conseguirmos ter todos os controles exibidos sem uso de scroll.

Vamos partir para a implementação dos novos controles, responsáveis pela exibição da fotografia do cliente e interação com a câmera e álbum de fotografias. Utilizaremos o `<ion-card>` para exibir a foto do cliente. Vamos querer que a foto apareça centralizada. Isso nos traz a necessidade de trabalharmos com estilos e já vimos no livro o uso de arquivos específicos para isso, os `scss` . Vamos então editar o arquivo `clientes-add-edit.page.scss` , que já havia sido criado com `generate` , com o código a seguir.

```
ion-card {
    margin-bottom: 0% !important;
    margin-top: 0% !important;
}
```

```
.parent {
    display: flex;
    height: 200px; /* Or whatever */
    background-color: #fff;
}

.child {
    width: 100px;  /* Or whatever */
    height: 100px; /* Or whatever */
    margin: auto;  /* Magic! */
}

.nopadding {
    margin-bottom: 0% !important;
    margin-top: 0% !important;
}
```

No início do código apresentado anteriormente, definimos um estilo específico para o controle `<ion-card>`, para garantirmos o menor espaço possível entre a imagem e os controles acima e abaixo dela. Você pode, e deve, testar a execução sem esta definição, para ver a diferença visual.

As classes `parent` e `child`, definidas também no código anterior, nos auxiliarão na exibição e centralização de nossa imagem. Não entrarei em detalhes específicos sobre `CSS`, por ser um conteúdo extenso, de certa maneira complexo e não ser o escopo de nosso livro. Entretanto, para quem desejar ser um desenvolvedor front-end, esse conhecimento é extremamente necessário.

Em nossa classe `ClientesAddEditPage`, no arquivo `clientes-add-edit.page.ts`, na mesma pasta do template, precisamos informar este novo arquivo criado, para que a renderização o considere. No código, falta-nos apenas `styleUrls`, mas trago o código completo a seguir.

```
@Component({
    templateUrl: './clientes-add-edit.page.html',
```

```
    styleUrls: ['./clientes-add-edit.page.scss']
})
```

Com o arquivo SCSS ajustado e referenciado, vamos ver na sequência o código necessário em nosso template para a exibição da foto e o botão com funcionalidades para capturá-la. Este código deverá ser inserido antes do `<ion-button>` para habilitar a alteração dos dados.

```
<ion-card>
    <ion-card-content>
        <div class="parent">
            <ion-img class="child" style="width: 150px; height: 150px;"
src="assets/imgs/icon_clientes.png"></ion-img>
        </div>
    </ion-card-content>
</ion-card>

<div *ngIf="modoDeEdicao">
    <ion-grid>
        <ion-row>
            <ion-col col-3 >
            </ion-col>

            <ion-col col-6>
                <ion-button class="nopadding"  shape="round"
color="primary" size="small" expand="block">
                    Capturar Foto
                </ion-button>
            </ion-col>

            <ion-col col-3>
            </ion-col>
        </ion-row>
    </ion-grid>
</div>
```

Em relação ao `<ion-card>`, a observação necessária recai sobre o `<div>` e `<ion-img>`, que referenciam as classes CSS implementadas

anteriormente. Estamos utilizando uma imagem básica apenas para termos um conteúdo exibido neste momento da implementação.

Após o `<ion-card>`, temos um `<div>` com a exibição condicional. Caso seja exibido, teremos o `<ion-button>` para capturar a foto renderizado. Note que fazemos uso do `<ion-grid>`, para que possamos centralizar o botão horizontalmente abaixo do `<ion-card>`. Também fazemos uso da classe de estilo `nopadding`. Vamos executar e ver como a aplicação está até aqui?

8.2 Interação com a câmera e o álbum de imagens

O processo de seleção de uma imagem para representar a foto do cliente se refere à possibilidade de o usuário selecionar uma imagem já existente na galeria de fotos, o álbum do dispositivo, ou então, capturar uma imagem diretamente da câmera. Veremos nesta seção estas duas funcionalidades para nosso aplicativo.

Vamos aproveitar que o capítulo é cheio de novidades e trazer mais um controle novo. O `Action Sheet`, comum nas aplicações para dispositivos móveis. Veja-o em execução na figura a seguir.

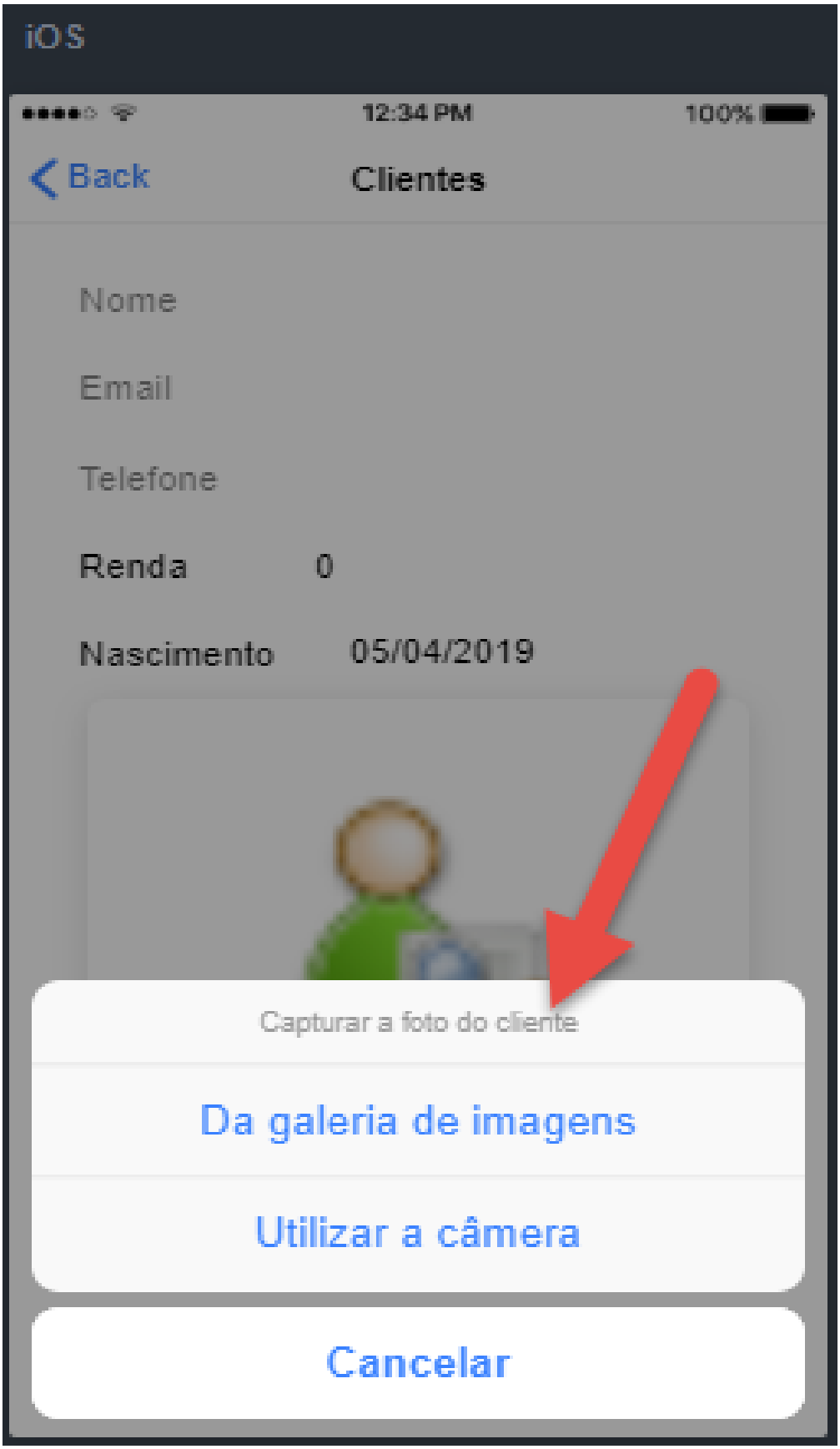

iOS
12:34 PM
100%
Back
Clientes
Nome
Email
Telefone
Renda 0
Nascimento 05/04/2019
Capturar a foto do cliente
Da galeria de imagens
Utilizar a câmera
Cancelar

Figura 8.2: Opções ao usuário com Action Sheet

Nossa ideia é que, ao usuário pressionar o botão para capturar uma imagem, seja oferecida a ele a possibilidade de escolher se a imagem virá da galeria de imagens, ou diretamente da câmera. Precisamos, inicialmente, atribuir ao controle em nosso template a invocação ao método que será responsável por disponibilizar estas opções. Veja o código atualizado a seguir. Observe a ligação com o evento `click`.

```
<ion-button class="nopadding"  shape="round" color="primary" size="small"
expand="block" (click)="capturarFoto()">
```

Com o código para o template adaptado, vamos implementar o método referenciado, o `capturarFoto()`. Veja-o na sequência, lembrando que ele deverá estar em nossa classe `ClientesAddEditPage`, no arquivo `clientes-add-edit.page.ts`, na mesma pasta do template. Observe o argumento `handler` para cada opção, que tem uma função anônima. Logo atribuíremos funcionalidade a ele.

```
async capturarFoto() {
    const actionSheet = await this.actionSheetController.create(
    {
        header: 'Capturar a foto do cliente',
        buttons: [{
            text: 'Da galeria de imagens',
            handler: () => {
            }
        },
        {
            text: 'Utilizar a câmera',
            handler: () => {
            }
        },
        {
            text: 'Cancelar',
            role: 'cancel'
        }
        ]
```

```
    });
    await actionSheet.present();
}
```

Observe que a primeira instrução atribui à `actionSheet` um objeto criado por `actionSheetController`. Veja o `this`, que nos informa que esse objeto pertence à nossa classe. Ocorre que ainda não temos o objeto. Precisamos injetá-lo em nossa classe. Para isso, vamos inserir a instrução a seguir no construtor. Será preciso importar `ActionSheetController` na classe.

```
private actionSheetController: ActionSheetController
```

Se você quiser, pode testar essa nova implementação em sua aplicação. Não temos ainda funcionalidades para as opções, pois o `handler()` delas, como dito, está sem comportamento.

Gostou do visual do `Action Sheet`? Mas vamos implementar a integração com a câmera e álbum de imagens? Você verá que, a princípio, é tudo muito simples. Faremos uso de componentes que realizam todo o trabalho pesado para nós. Dessa maneira, na pasta de seu projeto, no terminal, execute as seguintes instruções para a instalação dos plugins que serão necessários para esta atividade.

```
npm install @capacitor/camera
```

```
npm install @capacitor/filesystem
```

Com os downloads e instalações realizados, vamos agora codificar e começaremos criando um serviço específico para trabalhar com fotos, pois podemos precisar deste recurso em diversos pontos de nossa aplicação. Sendo assim, no terminal, digite a instrução a seguir.

```
ionic g service services/fotos
```

Vamos agora implementar um método que será responsável por ativar a câmera e então possibilitar ao usuário tirar uma foto. Veja na sequência o primeiro método a ser inserido nessa nossa nova

classe de serviço. Observe que temos um método assíncrono. A execução desta funcionalidade é simples, basta invocarmos o método `getPhoto()` do plugin `Camera`, do Capacitor, que instalamos há pouco.

```
async obterFoto() {
    const capturedPhoto = await Camera.getPhoto({
      resultType: CameraResultType.Uri,
      source: CameraSource.Camera,
      saveToGallery: true,
      quality: 100,
    });
}
```

O plugin `Camera` que instalamos, tem em `source` a possibilidade de se atribuir `CameraSource.Photos`, mas não tive sucesso com essa opção, pois acusa falta de permissões para escrita e leitura, embora a documentação diga que o plugin faz isso automaticamente. Sendo assim, utilizaremos depois outra estratégia para isso. Por enquanto, vamos trabalhar com a foto obtida da câmera. Observe também que estamos definindo `100%` para a qualidade final da foto, que o tipo para o retorno é `Uri`, mas existem outros, como a `base64`. Também estamos permitindo que a foto tirada seja registrada na galeria de fotos.

Para selecionarmos uma imagem, continuaremos com o mesmo plugin, mas invocando um método diferente, o `pickImages()`. Este método possibilita inclusive a seleção de mais de uma imagem e precisaremos pensar em tratar isso, ou assumir apenas a primeira imagem selecionada. Será interessante você pensar em sua solução para esta funcionalidade. Vamos então ao método, que deve ser implementado também em nosso `fotos.service.ts`. Veja-o na sequência. Observe `limit`, que funciona, pela documentação, apenas no iOS, o que nos auxilia para o problema comentado.

```
async escolherFoto() {
    const pickedPhoto: GalleryPhotos = await Camera.pickImages({
      quality: 100,
```

```
      limit: 1
    });
}
```

Precisamos agora invocar esses métodos quando o usuário selecionar o desejo de capturar uma foto da câmera, ou da galeria. Vamos realizar esta implementação no método `capturarFoto()`, na `arrow function` para os `handlers`. Lembra que falei delas? Veja as implementações a seguir, para galeria e para câmera, respectivamente. Observe que estamos fazendo uso de `this.fotosService`, o que leva à dedução de que precisamos injetar `FotosService` no construtor da classe do arquivo `clientes-add-edit.page.ts`.

```
handler: async () => {
  await this.fotosService.escolherFoto();
}

handler: async () => {
  await this.fotosService.obterFoto();
}
```

Com esta implementação já podemos realizar um teste em nossa aplicação e ver se conseguimos abrir a câmera para tirar uma foto e também vamos ver o comportamento para a galeria. Bora lá?

8.3 Armazenando no dispositivo a imagem capturada

Já temos a captura de nossa imagem funcionando. No último teste, vimos que selecionamos a imagem do álbum de imagens ou tiramos uma foto, mas não fizemos, ainda, nada mais que isso. Não temos a imagem exibida, pois ao selecionar a imagem não realizamos nenhum processo em que pudéssemos apontá-la para nosso componente. Temos algumas etapas a vencer e faremos isso aqui

nesta seção, por meio de métodos que criaremos e consumiremos, assim como adaptaremos alguns já implementados.

O primeiro método que implementaremos refere-se ao nome que daremos à imagem. Poderíamos pensar em dar a esta imagem o valor que temos para o `id` de cada cliente, pois ele é gerado pelo Firestore e é único. Mas como faríamos para quando estamos inserindo os dados de um cliente novo? Ele ainda não teria este valor. Podemos pensar em algumas possibilidades para esta situação. A primeira seria gerar o `id`, caso ele ainda não exista, mas isso quebraria toda a lógica que já temos para a inserção de um novo cliente, pois o `id` é nosso flag. Poderíamos também impedir a captura antes de o cliente ter o `id`. Ou seja, para capturar a foto do cliente, ele precisaria antes estar persistido. São duas possibilidades que atingem códigos que já temos implementados e funcionais. As dicas ficam e valem, no mínimo, para uma análise em situações futuras que possam precisar de uma imagem.

Nós tomaremos uma decisão mais simples, mais trivial. Manteremos tudo que já temos implementado e criaremos o arquivo com o nome ligado à data e hora do momento da captura. É mais trivial, mais simples, como dito, mas não atrapalha em nada o que já temos e o que venhamos a ter. É algo independente.

Parece desnecessário, mas vamos criar um método que realize este comportamento, promovendo uma coesão em nossa implementação. Veja este código na sequência. A implementação deve ser feita em nosso arquivo `fotos.service.ts`.

```
private _criarNomeArquivoImagem() {
    const d = new Date(),
        t = d.getTime(),
        novoNomeArquivo = t + '.jpg';
    return novoNomeArquivo;
}
```

Notou a simplicidade do método anterior? Obtemos a data por `Date()` e um valor numérico, que representa a data e hora universal

em milissegundos. Veja também que utilizamos uma única vez `const`, para declarar três variáveis. Ao final, retornamos o nome criado.

Toda imagem que obtivermos pela câmera ou recuperarmos do álbum de imagens estará persistida fisicamente no dispositivo, em um local que é de controle do sistema operacional, em conjunto com regras específicas para cada aplicativo nele instalado. Normalmente, este local é temporário, assim como o tempo de vida das imagens capturadas. Precisamos gravá-las também no dispositivo, em um local em que a imagem permanecerá no contexto da aplicação, até que nós a removamos, ou que o aplicativo seja removido. Veja o código para esta funcionalidade implementado na listagem a seguir, que deverá estar no `fotos.service.ts`.

```
private async _gravarFoto(caminho: string) {
    const nomeDoArquivo = this._criarNomeArquivoImagem();
    const dadosEmBase64 = await this._lerComoBase64(caminho);

    const arquivoGravado = await Filesystem.writeFile({
      path: nomeDoArquivo,
      data: dadosEmBase64,
      directory: Directory.Data
    });
}
```

Devido ao `Filesystem` e `Directory`, precisamores implementar o `import` a seguir.

```
import { Directory, Filesystem } from '@capacitor/filesystem';
```

Notou que na segunda linha do método temos a invocação ao `_lerComoBase64()`, que ainda não temos implementado? Pois é. De acordo com a documentação, há essa recomendação de trabalhar com uma imagem em uma string `Base64` e, para isso, precisamos converter o arquivo do caminho recebido pelo método para este tipo de dado. Veja então a implementação deste método no código a seguir, que deverá estar no `fotos.service.ts`.

```
private async _lerComoBase64(caminho: string) {
    const response = await fetch(caminho);
    const blob = await response.blob();

    return await this._converterBlobParaBase64(blob) as string;
}
```

O método `fetch()`, utilizado no código anterior, é um recurso que promove a leitura de um arquivo em um formato `Blob` (coleção de dados binários). Depois, a transformação da resposta a este método invoca o método `blob()`, que tem seu valor utilizado na invocação a outro método, o `_converterBlobParaBase64`, que tem sua implementação na sequência. Este método converte o blob em uma string com base 64, que é o que queremos.

```
_converterBlobParaBase64 = (blob: Blob) => new Promise((resolve, reject)
=> {
    const reader = new FileReader;
    reader.onerror = reject;
    reader.onload = () => {
      resolve(reader.result);
    };
    reader.readAsDataURL(blob);
});
```

Observe, no código anterior, o uso de `Promise`, `resolve` e `reject` para a manipulação do dado recebido e utilizado pelo `FileReader`.

Já temos toda a lógica para obtenção da foto, seja por câmera ou galeria, e a gravação desta imagem no contexto de nosso aplicativo. Precisamos agora pensar em como disponibilizar este arquivo para ser consumido pela nossa interface com o usuário para a página de clientes. Começaremos declarando uma variável pública para armazenar o caminho da imagem gravada. Insira o código a seguir logo na primeira linha da classe `FotosService`.

```
public caminhoParaFoto: string = '';
```

Agora, precisamos pensar em como atribuir o caminho para a imagem gravada a esta propriedade, mas isso é fácil. Vamos inserir a instrução a seguir como última linha do método `_gravarFoto()`.

```
this.caminhoParaFoto = arquivoGravado.uri;
```

E precisamos, ao final do `escolherFoto()` e `obterFoto()`, invocar o método `_gravarFoto()`, da maneira que temos na sequência.

```
this.caminhoParaFoto = arquivoGravado.uri;
```

Vamos agora trabalhar com o código para o template da página de cliente, pois precisamos ligar a imagem que está fixa no HTML como uma variável ligada ao código. Então, no `clientes-add-edit.page.ts`, logo no início da classe, declare a propridade a seguir. Veja que estamos inicializando ela com o valor para a situação inicial de renderização.

```
public caminhoParaFoto: string = 'assets/imgs/icon_clientes.png';
```

Na sequência, vamos ajustar nosso HTML, tal qual apresento no código que se segue. Note o que estamos atribuindo à `src`. Já vimos essa ligação antes.

```
<ion-img class="child" style="width: 150px; height: 150px;" src=
{{caminhoParaFoto}}></ion-img>
```

Precisamos ajustar esta propriedade quando houver a seleção de uma imagem e faremos isso implementando a instrução que se segue nos `handlers` do action sheet. Note a necessidade de invocar o método `convertFileSrc()`, do `Capacitor`, que precisará ser importado. Isso se faz necessário para que a URL do arquivo possa ser compreendida pelo Ionic e a renderização da imagem seja possível. Também poderíamos ter os métodos invocados nos handlers retornando este valor e então realizaríamos a atribuição diretamente, mas aqui optei pela primeira maneira.

```
this.caminhoParaFoto =
Capacitor.convertFileSrc(this.fotosService.caminhoParaFoto);
```

Agora podemos testar nossa aplicação. Tire uma foto e selecione uma da galeria. Veja se elas aparecem. Bora lá?

Temos o caminho para a imagem do cliente, mas não temos este dado ligado ao nosso modelo de negócio, a classe `Cliente` e precisamos ajustar isso. Nosso primeiro passo é, em nossa interface `Cliente`, localizada em `/src/app/models/cliente.model.ts`, inserir nossa nova propriedade ao final do código, após `nascimento`, como no código a seguir, que também traz instruções para o final dos converters relacionados ao Firestore.

```
// interface Cliente
foto: string;

// toFirestore
foto: cliente.foto,

// fromFirestore
foto: data.foto,
```

Temos agora alguns pontos para associarmos nossa nova propriedade. O primeiro se dará no método `submit()`, antes da invocação ao `if()` que invoca o `create()` ou `update()`, inserindo a instrução a seguir, que não é novidade para nós. Apenas atribuímos a um controle de nosso formulário um valor, que neste caso é a imagem que temos como selecionada para o cliente atual. Observe que trato a não seleção da imagem. Você poderia pensar em ter a imagem como algo obrigatório também.

```
if (this.fotosService.caminhoParaFoto != '') {

this.clienteForm.controls.foto.setValue(this.fotosService.caminhoParaFoto)
;
} else {
  this.clienteForm.controls.foto.setValue('');
}
```

Precisamos também adaptar o método `create()`, de `FotosService` para que tenha a atribuição da foto. Basta inserirmos a instrução a

seguir nos dados enviados para `setDoc(doc(clienteRef), {})`.

```
foto: cliente.foto,
```

Mas e quando estivermos acessando um cliente já cadastrado, que pode ter uma foto já informada? Precisamos dela exibida. Vamos fazer isso na declaração do objeto `cliente`, de nossa classe. Veja o código a seguir, que faz parte do `ionViewWillEnter()`. Veja a nova implementação tanto para o `if`, como para o `else`, que tratam a recuperação de um cliente já existente, quanto a de um novo, respectivamente.

```
if (id !== '-1') {
    this.cliente = await this.clientesService.getById(id);
    this.caminhoParaFoto = this.cliente.foto;
} else {
    this.cliente = {clienteid: '', nome: '', email: '', telefone: '',
renda: 0.00, nascimento: new Date(),
        foto: this.caminhoParaFoto };
    this.modoDeEdicao = true;
}
```

Para concluirmos esta etapa, no mesmo método da listagem anterior, onde declaramos nosso objeto responsável pelo formulário, precisamos também adotar a nova propriedade `foto`. Para facilitar, trago na sequência a declaração completa do objeto.

```
this.clienteForm = this.formBuilder.group({
    clienteid: [this.cliente.clienteid],
    nome: [this.cliente.nome, Validators.required],
    email: [this.cliente.email, Validators.required],
    telefone: [this.cliente.telefone, Validators.required],
    renda: [this.cliente.renda, Validators.required],
    nascimento: [this.cliente.nascimento, Validators.required],
    nascimentoForm: [{ value:
this.cliente.nascimento.toLocaleDateString(), disabled: !this.modoDeEdicao
}, Validators.required],
    foto: [this.cliente.foto]
});
```

Com toda esta grande etapa vencida, podemos testar nossa aplicação. Recupere uma imagem do álbum de fotos e da câmera. Para o Android isso pode ser feito no emulador. Para o iPhone, precisamos de um dispositivo.

8.4 Visualização das imagens dos clientes na listagem

Já temos a inserção e alteração funcionando de maneira satisfatória em relação ao objetivo que tínhamos. Agora, precisamos adaptar nossa listagem, para que a foto de cada cliente seja exibida também na listagem, e não apenas no momento da gravação.

Vimos, na etapa anterior, que precisamos transformar o endereço para a foto para uma maneira que o Ionic consiga renderizar. Como estamos armazenando o caminho físico para a imagem, precisaremos desta transformação também na listagem dos clientes e podemos ver um método com essa responsabilidade na sequência. É no arquivo `clientes-listagem.page.ts`, ok? Veja que estamos novamente utilizando uma string para a imagem a ser exibida na ausência de uma foto. Poderíamos trabalhar isso em uma constante e reutilizá-la. Será preciso importar `Capacitor`.

```
caminhoFotoParaListagem(foto: string) {
    if (foto != '') {
      return Capacitor.convertFileSrc(foto);
    } else {
      return 'assets/imgs/icon_clientes.png';
    }
}
```

Agora vamos para a camada de apresentação, o HTML. Bora adaptar nosso `<ion-item>` para o código a seguir. Algumas adaptações também foram feitas buscando um melhor visual. Este código está em `clientes-listagem.page.html`.

```
<ion-item routerDirection="forward" [routerLink]="['/', 'clientes-add-
edit', cliente.clienteid]">
    <ion-thumbnail slot="start">
      <ion-img [src]="caminhoFotoParaListagem(cliente.foto)"></ion-img>
    </ion-thumbnail>
    <ion-grid>
      <ion-row>
        <ion-label>
          <p style="font-size: larger; font-weight: bold;">
{{cliente.nome}}</p>
        </ion-label>
      </ion-row>
      <ion-row>
        <ion-col col-12>
          <ion-label>
            <p style="font-weight: 400; text-align: right">Telefone:
{{cliente.telefone}}</p>
          </ion-label>
        </ion-col>
      </ion-row>
    </ion-grid>
</ion-item>
```

Observou o uso do `<ion-thumbnail>` para nossa imagem? Verificou a ligação de `[src]` com `caminhoFotoParaListagem(cliente.foto)` ? Para os dados de cada cliente, estamos utilizando um `<ion-grid>` , componente já conhecido por nós. Você pode uma vez mais testar sua aplicação, e verificará que a imagem que selecionamos para o cliente, em nosso último teste, aparece na listagem, ao lado dos dados do cliente que inserimos.

8.5 Remoção da imagem física do dispositivo

Mas e na remoção do cliente? A imagem ficará ainda no dispositivo. Não queremos isso, e nem nosso usuário, correto? Vamos então adaptar um pouco nosso código do método `removerCliente()` . Para

relembrar, trago toda a implementação na sequência, para ajudar na compreensão. Esta implementação é no arquivo TypeScript do template de listagem. Importações para o `Filesystem` e `Directory` serão necessárias.

```
async removerCliente(cliente: Cliente) {
    try {
      const successFunction = async () => {
        this.clientesService.removeById(cliente.clienteid);

        const caminho: string = cliente.foto.substr(0,
cliente.foto.lastIndexOf('/') + 1);
        const nomeArquivo: string =
cliente.foto.substr(cliente.foto.lastIndexOf('/') + 1,
(cliente.foto.length - caminho.length));
        try {
          if (cliente.foto != '') {
            await Filesystem.deleteFile({
              path: nomeArquivo,
              directory: Directory.Data,
            });
          }
          this.toastService.presentToast('Cliente removido com sucesso',
3000, 'top');
          this.slidingList.closeSlidingItems();
        } catch (e) {
          await this.alertService.presentAlert('Falha', 'Remoção do
arquivo não foi executada', e, ['Ok']);
        }
        this.clientes = await this.clientesService.getAll();
      };
      await this.alertService.presentConfirm('Remover Cliente', 'Confirma
remoção?', successFunction);
    } catch (e) {
      await this.alertService.presentAlert('Falha', 'Remoção não foi
executada', e, ['Ok']);
    }
}
```

Notou, na declaração de `successFunction`, que após a invocação do `removeById()` declaramos duas variáveis? Elas são necessárias para termos o nome do arquivo. Poderíamos pensar em um método específico para isso. Ainda é possível usarmos o `LoadingController`, caso você julgue necessário, para o usuário não ter a impressão de congelamento da aplicação.

Imagine que o usuário selecionou uma imagem, na inserção ou alteração de um cliente e, em vez de gravar, ele cancela o processo. O que acontece com a imagem capturada na inserção? Ela precisa ser eliminada. E se ele cancela em uma alteração? É preciso manter a anterior e eliminar a nova. E na troca de uma imagem por outra? Fica o desafio.

8.6 Vamos manter a foto também na nuvem

Sabemos que o espaço físico disponível em um dispositivo móvel, dependendo do modelo, fabricante, plataforma, é algo caro e precisamos otimizar este uso. Vimos no capítulo anterior o uso do Firestore para persistirmos nossos dados na nuvem. Por que não termos, então, nossa foto também na nuvem? Utilizaremos mais um produto do Firebase do Google, o `Storage`.

Precisamos preparar este serviço para nossa aplicação no console do Firebase. Desta maneira, com seu acesso realizado, clique na opção chamada `Storage`, que normalmente está em um menu ao lado esquerdo de sua tela. Opto por não apresentar uma imagem da console do Firebase por ser uma aplicação web e poder mudar facilmente.

Ao acessar pela primeira vez a opção Storage, algumas opções orientativas serão oferecidas, recomendo que você as leia. Podemos clicar no botão de `Primeiros passos` para começarmos a configuração deste serviço para nosso aplicativo.

Será apresentado a você um script que registrará as regras para acesso de leitura e escrita para este serviço. Como estamos em fase de desenvolvimento e testes e ainda não temos autenticação e muito menos autorização, vamos alterar a instrução responsável por isso no código. De início, clique na opção de confirmação.

Com este processo concluído, você verá uma mensagem semelhante a `As regras padrão de segurança exigem que os usuários se autentiquem`. É o que comentamos no parágrafo anterior. Clique na opção `Regras` no topo da página do console. Substitua a instrução de permissão (`allow`) pela que está a seguir, que permite acesso a todos que tenham as credenciais da aplicação.

```
allow read, write: if true == true;
```

Ao informar o código anterior, uma mensagem de opção de publicação da alteração, ou descarte dela, é imediatamente apresentada. Confirme a publicação. Podemos voltar à relação de arquivos, clicando na opção `Arquivos`.

Lembre-se de que para acessar o Storage, é preciso ter toda a configuração de acesso ao Firebase, tal qual já temos no arquivo `credentials.js`.

Teremos alterações de código e novas implementações. Para a técnica que adotaremos, que será enviar a imagem apenas na gravação do cliente, precisaremos ter um método responsável por essa funcionalidade. Precisaremos obter o nome do arquivo a ser enviado e vimos em códigos anteriores como fazer isso, mas agora faremos um método específico para isso, para que você possa pensar em reúso. Veja este código a seguir a ser implementado no `clientes-add-edit.page.ts`.

```
obterNomeDoArquivo() {
    return
this.fotosService.caminhoParaFoto.substr(this.fotosService.caminhoParaFoto
.lastIndexOf('/') + 1);
}
```

No mesmo arquivo, implementaremos agora o método responsável pelo envio do arquivo, que é um upload. Veja este código a seguir. Veja que invocamos `getStorage()` enviando a aplicação por meio da variável `this._fireStore`, que deve ser injetada no servidor. Com o Storage obtido, é obtida uma referência a ele, onde definimos uma pasta, seguida do nome do arquivo. Com tudo pronto, é invocado então o método `uploadBytes()`, que recebe a referência para o Storage e o Blob retornado pelo método `lerComoBlob()`, que é o arquivo a ser enviado. Veja o retorno do método, que invoca `getDownloadURL()`, que conterá o caminho até o arquivo armazenado no Storage.

```
async uploadFile() {
    const storage: FirebaseStorage = getStorage(this._fireStore.app,
this._fireStore.app.options.storageBucket);
    const storageReference = storageRef(storage, "fotosClientes/" +
this.obterNomeDoArquivo());

    await uploadBytes(storageReference, await
this.photoService.lerComoBlob(this.caminhoParaFoto));
    return await getDownloadURL(storageReference);
}
```

Os imports a seguir serão necessários para o código anterior, além da implementação do `lerComoBlob()`, em `FotosService`.

```
// import em clientes-add-edit.page.ts
import { getStorage, uploadBytes, ref as storageRef, getDownloadURL,
FirebaseStorage } from 'firebase/storage';

// método em FotosService
async lerComoBlob(caminho: string) {
    const response = await fetch(caminho);
    return await response.blob();
}
```

Precisamos agora invocar este método antes do `if()`, que é responsável pela invocação ao `create()` e `update()`. Veja a instrução

no código seguinte, seguida da responsável pelo `setValue()` caso haja uma imagem selecionada.

```
const urlArquivoEnviado = await this.uploadFile();

this.clienteForm.controls.foto.setValue(urlArquivoEnviado);
```

Vamos testar a aplicação? Insira um cliente, com foto, ou altere um já existente e grave-o. Depois, vá ao Storage na console do Firebase e veja lá o arquivo enviado.

Temos os dados e as fotos de nossos clientes na nuvem. Entretanto, em nossa listagem de clientes, precisamos ajustar o bind para o que temos no código a seguir. Veja que retiramos a invocação à função e exibimos direto a URL para o Storage.

```
<ion-img [src]="cliente.foto"></ion-img>
```

Precisamos ajustar ainda uma situação. Nós temos fisicamente o arquivo da imagem armazenado em nosso dispositivo e não precisamos mais dele, pois o enviamos para a nuvem. Vamos então removê-lo tão logo o upload seja concluído. Veja na sequência o método que precisamos implementar logo abaixo do `uploadFile()`. Imports serão necessários.

```
async removerFotoLocal() {
    if (this.fotosService.caminhoParaFoto != '') {
      await Filesystem.deleteFile({
        path: this.obterNomeDoArquivo(),
        directory: Directory.Data,
      });
    }
}
```

Agora, no `submit()`, logo após a invocação ao `uploadFile()`, invoque este nosso novo método, tal qual apresento na sequência. Após isso, realize novo teste.

```
await this.removerFotoLocal();
```

8.7 A remoção da imagem no Storage

Você experimentou, no teste anterior, trocar uma imagem de um cliente que já tenha uma foto? Funciona, certo? Mas você chegou a verificar no Storage que a imagem anterior não foi eliminada? Nós vamos, nesta seção, implementar o necessário para realizarmos este processo de remoção de imagem na listagem de clientes, que já tem a remoção dele implementada.

No momento, o código que temos para a remoção refere-se a remover um arquivo físico, e vimos que o esforço para recuperar o nome do arquivo na URL é pequeno. Já com o Storage, o link retornado após o download já é mais complexo e não depende de nós para gerá-lo, o que nos faz pensar que ele pode mudar e nossa lógica deixar de funcionar em algum momento. Isso nos remete a persistir o nome do arquivo individualmente, pois é o que enviamos para o upload. Vamos ajustar nosso modelo para possuir uma nova propriedade, tal qual o código a seguir, no arquivo `cliente.model.ts`.

```
// Interface Cliente
nomeDaFoto: string;

// toFirestore
nomeDaFoto: cliente.nomeDaFoto,

// fromFirestore
nomeDaFoto: data.nomeDaFoto
```

Precisaremos agora popular esta nova propriedade e faremos isso no arquivo `clientes-add-edit.page.ts`. A primeira alteração será no `ionViewWillEnter()`, onde, no `else`, inseriremos essa nova propriedade ao final da atribuição à `this.cliente`, tal qual o código a seguir.

```
nomeDaFoto: this.obterNomeDoArquivo()
```

Depois, na atribuição de `this.clienteForm`, vamos inserir também a atribuição para esta nova propriedade.

```
nomeDaFoto: [this.obterNomeDoArquivo()]
```

Precisaremos ajustar em `clientes.service.ts`, no método `create()`, a declaração da nova propriedade, tal qual como segue.

```
nomeDaFoto: cliente.nomeDaFoto,
```

Por fim, na atribuição da URL do arquivo enviado, dentro do `if` do `submit()`, precisamos também inserir a instrução a seguir.

```
this.clienteForm.controls.nomeDaFoto.setValue(this.obterNomeDoArquivo());
```

Como realizamos alterações em nosso modelo de negócio, não conseguiremos remover a imagem que já está lá no Storage. Precisaremos inserir um novo Cliente e então trabalharmos a remoção, que começa com a implementação do método a seguir, na classe `ClientesService`. Veja que recuperamos um objeto `FirebaseStorage`, enviando nossa aplicação firebase e o valor para `Storage Bucket`, que temos registrado na `credentials.ts`. Com este objeto, recuperamos uma referência para a imagem que desejamos remover e então a removemos por meio da invocação a `deleteObject()`.

```
async removerImagem(nomeImagem) {
    try {
        const storage: FirebaseStorage = getStorage(this._fireStore.app,
this._fireStore.app.options.storageBucket);
        const storageReference = storageRef(storage, "fotosClientes/" +
nomeImagem);
        await deleteObject(storageReference);
    } catch (e) {
        console.log(e);
    }
}
```

Em relação ao `storageRef`, que temos no código, precisamos importá-lo como segue na listagem, que já traz todos imports necessários.

```
import { deleteObject, FirebaseStorage, getStorage, ref as storageRef }
from 'firebase/storage';
```

Nós já temos o upload da foto e a remoção dela. Agora, um desafio para você. Aplique uma lógica para que quando ocorrer uma alteração nos dados do cliente e também na imagem já registrada, a nova imagem seja inserida e a antiga removida. Outro ponto importante para este desafio: quando capturamos uma imagem, se não gravarmos, ela continuará no disco. Pense em como pode removê-la caso o usuário desista da operação, seja inserção ou remoção.

Mas onde invocaremos o método que acabamos de implementar? Nós precisamos apenas substituir a remoção que temos, que é em relação ao arquivo físico, para invocar este método, tal qual trago na sequência.

```
await this.clientesService.removerImagem(cliente.nomeDaFoto);
```

Bora testar inserir, alterar e remover? Comprove tudo no console do Firebase.

Conclusão

Trabalhamos aqui a captura de imagens, tanto do álbum de fotografias como utilizando a câmera do dispositivo. Gravamos a imagem capturada no próprio dispositivo, depois vimos a importância de termos as imagens na nuvem e fizemos isso por meio do `Storage` do `Firebase` do Google. Processo relativamente simples. Foi realmente um capítulo muito importante para o desenvolvimento de aplicações para dispositivos móveis conectadas à nuvem.

No próximo capítulo trabalharemos a autenticação de nosso usuário em nossa aplicação e consequentemente no Firebase, pois da maneira que temos a implementação, o acesso aos nossos recursos está público, não temos restrições.

Capítulo 9
Autenticação e criação de usuários no Firebase e uso de Tabs

Implementamos até o capítulo anterior o consumo de dois serviços do Firebase: o Firestore, para persistência remota de dados e o Storage, para armazenar fotos de nossos clientes. Até aí tudo bem, porém, não temos nenhum controle para limitar o acesso apenas a usuários de nossa aplicação.

Poderíamos pensar em desenvolver toda uma arquitetura para controle de usuários e validá-lo sempre para que estes dados possam ser acessados. Entretanto, o Firebase também nos oferece um serviço de autenticação muito bom e faremos uso dele neste capítulo.

Como um último recurso para navegação entre páginas para o usuário, veremos o `Tabs`, um controle que exibe na tela de seu dispositivo alguns botões, pelos quais o usuário terá acesso a páginas da aplicação. É outra maneira de navegação, além do Sidemenu que já conhecemos.

9.1 Preparação do projeto para a autenticação

Neste capítulo, vou optar por criarmos um projeto novo e nele implementar o que vamos utilizar. Para facilitar, veremos todos os passos aqui que serão necessários para nosso novo projeto.

Acesse o terminal, na pasta que armazena seus projetos e nela digite a instrução a seguir. Após a conclusão, execute a segunda instrução apontada.

```
ionic start capitulo09 blank --type=angular --no-git

// dentro da pasta criada
npm run build
```

Como faremos uso do Firebase no projeto, precisamos instalar nele as dependências necessárias. Desta maneira, na pasta de seu projeto, digite a seguinte instrução e aguarde a instalação concluir.

```
npm install @angular/fire firebase
```

Já temos nossa aplicação criada no Firebase, precisamos agora apenas configurar o processo de autenticação. Para isso, no menu de desenvolvimento de sua aplicação, lá no Console Firebase, clique na opção `Authentication` e, quando a nova página for exibida, no menu superior dela, clique em `Método de login` (Sign in method).

Com a nova página exibida, será possível identificar uma boa variedade de mecanismos de controle de autenticação a que o Firebase está integrado. Para nossa aplicação inicial deste capítulo, nós trabalharemos com `E-mail/senha` . Basta então clicarmos no ícone que aparece ao lado direito da opção quando colocamos o ponteiro do mouse sobre ela.

Quando formos ativar este método, uma janela será exibida, solicitando sua confirmação do que deseja ativar. Nós queremos apenas ativar o método de autenticação por email e senha. Habilite apenas esta opção. Veja a figura a seguir para lhe orientar, lembrando que ela pode mudar dependendo sempre da atualização da ferramenta do Google.

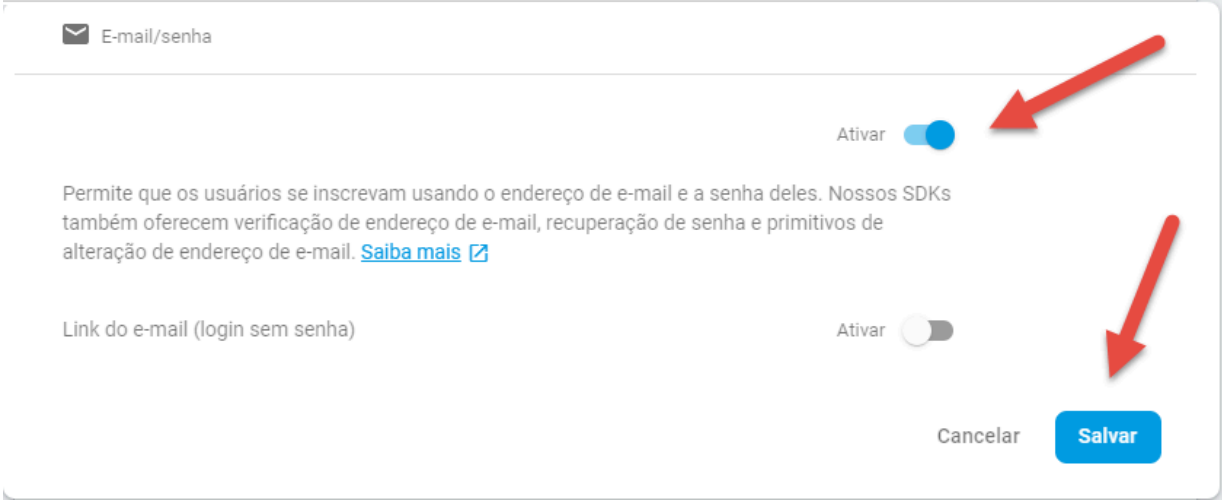


Figura 9.1: Ativando autenticação por e-mail/senha

Precisaremos agora trazer para este nosso novo projeto as configurações de ambiente para conexão com o Firebase. As mesmas que utilizamos nos capítulos anteriores. Elas estão em um arquivo chamado `credentials.ts`, na pasta `/src/app`. Você pode copiar o arquivo para o mesmo caminho no novo projeto ou criar o arquivo e colar as configurações. Para relembrar, trago na sequência as configurações, que você deve adaptar para seus dados.

```
export const environment = {
    production: false,
    useEmulators: true,
    firebaseConfig: {
        apiKey: "AIzaSyDmlXzNpTayhN4jCWDkgAi-qFWvxtMSKfs",
        authDomain: "ionic5-c0dc4.firebaseapp.com",
        databaseURL: "https://ionicccfb.firebaseio.com",
        projectId: "ionic5-c0dc4",
        storageBucket: "ionic5-c0dc4.appspot.com",
        messagingSenderId: "624846692713",
        appId: "1:624846692713:web:40d3e49ffdeb3940939df8"
    },
};
```

9.2 O login para acessar a aplicação

Nosso primeiro passo para a implementação de nossa autenticação será a criação de nossa visão de login, pela qual o usuário informará um email e uma senha, que serão validadas e então o acesso poderá ser liberado, ou não. No terminal, digite a seguinte instrução.

```
ionic generate page login
```

Com a página criada, podemos eliminar o arquivo `login.page.spec.ts`, pois é para testes e não estamos vendo isso. Para finalizar o processo e termos nossa página configurada como de início, vamos alterar o `redirectTo` do `path: ''` para `login`.

Vamos trabalhar agora nossa classe que renderizará o template para o usuário, a `login.page.ts`, que tem a primeira parte do código apresentada na sequência, que você deve implementar em seu projeto.

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormBuilder, Validators, FormControl } from
'@angular/forms';
import { NavController } from '@ionic/angular';

@Component({
  selector: 'app-login',
  templateUrl: './login.page.html',
  styleUrls: ['./login.page.scss'],
})
export class LoginPage implements OnInit {
  validationsForm: FormGroup;
  errorMessage: string;

  validationMessages = {
    email: [
      { type: 'required', message: 'Informe o E-Mail' },
      { type: 'pattern', message: 'Informe um email válido' }
    ],
    password: [
```

```
      { type: 'required', message: 'Informe a senha' },
      { type: 'minlength', message: 'A senha precisa ter ao menos 6
caracteres' }
    ]
  };
}
```

Vamos à segunda parte do código? Veja-a na sequência. Inseri o cabeçalho da classe apenas para que a indentação ficasse correta.

```
export class LoginPage implements OnInit {

  // Código apresentado anteriormente

  constructor(
    private formBuilder: FormBuilder
  ) { }

  ngOnInit() {
    this.validationsForm = this.formBuilder.group({
      email: new FormControl('', Validators.compose([
        Validators.required,
        Validators.pattern('^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+.[a-zA-Z0-
9-.]+$')
      ])),
      password: new FormControl('', Validators.compose([
        Validators.minLength(6),
        Validators.required
      ])),
    });
  }
}
```

Temos a injeção de um `formBuilder` no construtor, que já estamos utilizando no `ngOnInit()` para a criação dos controles no `Form`. Verifique as regras que devem ser aplicadas nos controles `email` e `password`. Com esta implementação realizada, podemos seguir para nosso template, o `login.page.html`. Veja atenciosamente o código na sequência.

```
<ion-header>
  <ion-toolbar color="primary">
    <ion-title>Login</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content padding>
  <form class="form" [formGroup]="validationsForm">
    <ion-item>
        <ion-label  position="floating" color="primary">Email</ion-label>
        <ion-input type="text" formControlName="email"></ion-input>
    </ion-item>

    <div>
      <ng-container *ngFor="let validation of validationMessages.email">
        <div class="validation-errors"
*ngIf="validationsForm.get('email').hasError(validation.type) &&
(validationsForm.get('email').dirty ||
validationsForm.get('email').touched)">
          {{ validation.message }}
        </div>
      </ng-container>
    </div>

    <ion-item>
      <ion-label  position="floating" color="primary">Password</ion-label>
      <ion-input type="password" formControlName="password" class="form-
controll" required></ion-input>
    </ion-item>
    <div class="validation-errors">
      <ng-container *ngFor="let validation of
validationMessages.password">
        <div class="error-message"
*ngIf="validationsForm.get('password').hasError(validation.type) &&
(validationsForm.get('password').dirty ||
validationsForm.get('password').touched)">
          {{ validation.message }}
        </div>
      </ng-container>
```

```
    </div>

    <div style="text-align: center" padding>
      <ion-button  class="submit-btn" type="submit"
[disabled]="!validationsForm.valid">Acessar</ion-button>
    </div>

    <div style="text-align: center">
      <label class="validation-errors">{{errorMessage}}</label>
    </div>
  </form>
  <p style="text-align: center">
    Não tem conta ainda? <a>Crie uma conta</a>
  </p>
</ion-content>
```

Precisamos ajustar o arquivo de estilos, com o código a seguir.

```
.validation-errors {
    text-align: center;
    color: red;;
}
```

Para que possamos testar nossa aplicação e vermos nosso formulário de login renderizado, precisamos importar `ReactiveFormsModule` em nosso arquivo `app.module.ts` e `login.module.ts` . Veja o código a seguir, com a orientação de onde inserir.

```
@NgModule({
  imports: [
    // Ocultado
    ReactiveFormsModule
  ],
  // Ocultado
})
```

A figura a seguir mostra o login em nosso pseudoemulador. Você pode digitar valores nos campos para validar as mensagens de erro.

Caso você tenha esquecido, trago antes da figura a instrução para executar a aplicação.

```
ionic capacitor run android -l --external
```

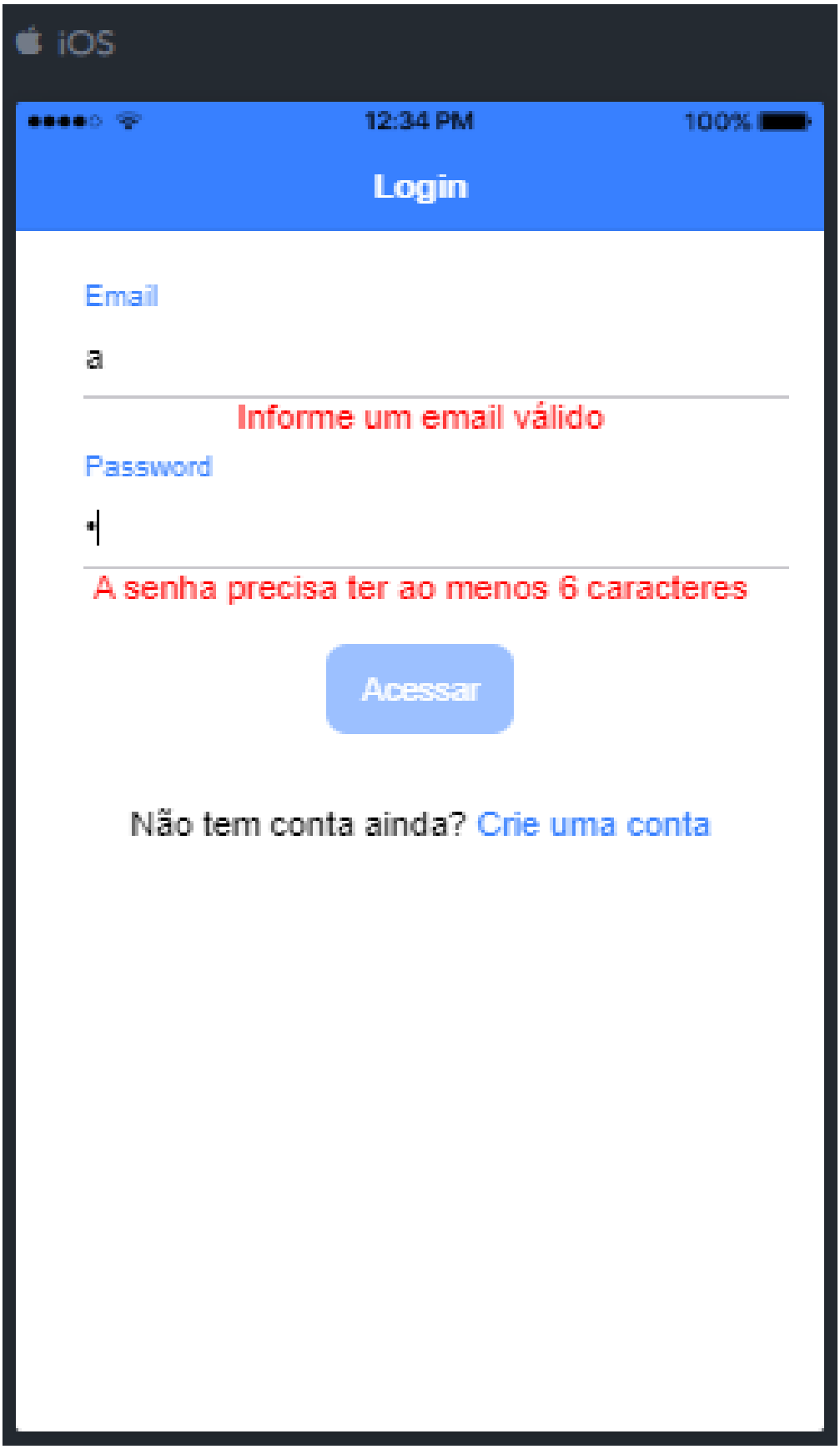

iOS
12:34 PM
100%
Login
Email
a
Informe um email válido
Password
A senha precisa ter ao menos 6 caracteres
Acessar
Não tem conta ainda? Crie uma conta

Figura 9.2: Página de login

Vamos agora implementar a funcionalidade que validará os dados de email e senha do usuário para autenticação no Firebase. Vamos trazer para cá novamente o uso de serviços, tal qual vínhamos fazendo para a persistência de dados. Acesse o terminal e digite a instrução a seguir.

```
ionic g service services/autenticacao
```

Vamos agora a algumas implementações para este nosso serviço. A primeira delas é importarmos recursos do Firebase para a classe. Vamos então inserir as instruções a seguir no início do arquivo.

```
import { getAuth, signInWithEmailAndPassword,
createUserWithEmailAndPassword } from "@firebase/auth";
import { initializeApp } from '@firebase/app';
import { environment } from '../credentials';
```

Na sequência, precisamos implementar nosso método responsável pela tentativa de autenticação do usuário no Firebase. Veja-o na sequência, com algumas explicações logo após, mas já verifique a primeira instrução, invocando a inicialização do Firebase antes de tentar o login.

```
async login(value) {
    initializeApp(environment.firebaseConfig);

    const auth = getAuth();
    try {
      return await signInWithEmailAndPassword(auth, value.email,
value.password);
    } catch (error) {
      throw new Error(error.message);
    }
}
```

Percebeu que não temos nada de complexo no método? Receberemos um objeto, que terá `email` e `password` como propriedades. Enviamos estes valores para um método do Firebase

chamado `signInWithEmailAndPassword()`, enviando o retornado por `auth()` como primeiro argumento. Em caso de sucesso, um objeto de `UserCredential` será retornado, caso contrário um `error`.

Seguindo o fluxo para a implementação, precisamos invocar este método em nossa classe que renderizará o template, a `LoginPage`, que está no arquivo `login.page.ts`. Precisamos injetar nosso serviço em nossa classe, inserindo a instrução a seguir em nosso construtor.

```
private autenticacaoService: AutenticacaoService
```

Precisaremos implementar também um método em nossa classe da página, que será invocado pelo template. Veja este método na sequência.

```
async login(value) {
    try {
      await this.autenticacaoService.login(value)
      this.errorMessage = 'Autenticado';
    } catch (error) {
      this.errorMessage = error.message;
    }
}
```

No código anterior, invocamos o método `login` de nosso serviço, enviando a ele um valor que será recebido por este método. Em caso de sucesso, será escrito `Autenticado` no controle que exibe erros de conexão, apenas para teste. Depois trabalharemos o redirecionamento para uma página da aplicação. Em caso de erro, a mensagem recuperada pelo erro será renderizada. Será uma mensagem padrão do Firebase. Você pode trabalhar essas mensagens e exibi-las de acordo com sua necessidade.

Precisamos possibilitar agora que nosso usuário requisite este método quando interagir com o botão de acesso à aplicação. Vamos adaptar nosso elemento `<form>` do template, tal qual é apresentado na sequência, com o *binding* de evento para `ngSubmit`.

```
<form class="form" [formGroup]="validationsForm"
(ngSubmit)="login(validationsForm.value)">
```

Vamos testar novamente nossa aplicação? Tente informar um email e senha e então clique no botão acessar. Notou o erro? No link https://firebase.google.com/docs/reference/js/firebase.auth.Auth.html#signinwithemailandpassword você pode verificar, quando quiser, algumas orientações sobre como identificar os erros possíveis na invocação do método `signInWithEmailAndPassword()`.

Com o que temos até agora, é preciso que você crie um usuário em seu Firebase Console. Basta acessar o menu `Authentication` depois na opção `Usuários` e então em `Adicionar Usuário`. Depois disso, teste com as credenciais válidas e veja a mensagem de sucesso abaixo do botão.

9.3 A aplicação acessível por Tabs

Quando nosso usuário é autenticado, o correto é que ele seja direcionado a uma página da aplicação. Pode ser uma página apenas, pode ser o Sidebar Menu que implementamos anteriormente, ou ainda, uma página que possua outras páginas, acessíveis via `Tabs`, que é o foco desta seção.

Começaremos com a criação da página inicial de nossa aplicação, que será exibida após a autenticação correta por parte do usuário e a chamaremos de `home`. Mas, como criamos nosso projeto do início, por padrão o Ionic já cria a página `home`. Vamos substituir o código que temos no template pelo que tem a seguir.

```
<ion-tabs>
  <ion-tab-bar slot="bottom">
    <ion-tab-button tab="pecas">
      <ion-icon name="pricetags"></ion-icon>
      <ion-label>Peças</ion-label>
```

```
    </ion-tab-button>

    <ion-tab-button tab="clientes">
      <ion-icon name="person"></ion-icon>
      <ion-label>Clientes</ion-label>
    </ion-tab-button>

    <ion-tab-button tab="atendimentos">
      <ion-icon name="car"></ion-icon>
      <ion-label>Atendimentos</ion-label>
    </ion-tab-button>

    <ion-tab-button (click)="logout()">
      <ion-icon name="log-out"></ion-icon>
      <ion-label>Sair</ion-label>
    </ion-tab-button>
  </ion-tab-bar>
</ion-tabs>
```

A página `home` será a página que abrigará o menu com `Tabs` acessíveis para o usuário. Essa definição começa com `<ion-tabs>`, que contém a barra de opções que será exibida, configurada pelo componente `<ion-tab-bar>`, que é composto de `<ion-tab-button>`, representando cada opção acessível.

Em `<ion-tab-button>` temos `tab`, que define um nome que será utilizado como rota para o Angular poder redirecionar a exibição da página correta dentro da página `home`. O último `<ion-tab-button>` não tem um `tab`, mas está ligado ao evento `click`, que invocará um método chamado `logout()`.

Com estas informações, podemos implementar agora nossa classe ligada ao template `home`. Veja o código dela na sequência.

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { AutenticacaoService } from '../services/autenticacao.service';
import { NavController, IonTabs } from '@ionic/angular';

@Component({
```

```
  selector: 'app-home',
  templateUrl: './home.page.html',
  styleUrls: ['./home.page.scss'],
})
export class HomePage implements OnInit {

  constructor(
    private autenticacaoService: AutenticacaoService,
    private navCtrl: NavController
  ) { }

  ngOnInit() { }

  logout() {
    this.autenticacaoService.logout()
    .then(res => {
      console.log(res);
      this.navCtrl.navigateBack('');
    })
    .catch(error => {
      console.log(error);
    });
  }
}
```

No método `logout()` da listagem anterior, apenas invocamos o método `logout()` que precisamos implementar em nosso `AutenticacaoService`, que você pode verificar na listagem a seguir.

```
async logout() {
    try {
      await getAuth().signOut();
    } catch (error) {
      throw new Error(error.message);
    }
}
```

Para o `logout()`, seguimos a mesma lógica adotada para o `login()`, mas de uma maneira um pouco diferente. Não enviamos o retorno

de `getAuth()` para ninguém, mas por meio de seu retorno invocamos `signOut()`. Seria possível verificar por meio da propriedade `currentUser` de `getAuth()` a ausência ou não de um usuário autenticado.

Mas como chegaremos em `home`? Nós já temos em `app-routing.module.ts` a rota a seguir, basta retornar `home` ao `redirectTo`, e `path: ''`.

Vamos agora rever nosso código de `login()` da página de `login`. Veja sua nova implementação na sequência. Temos só duas alterações, mas preferi trazê-lo todo.

```
async login(value) {
    this.autenticacaoService.login(value)
    .then(res => {
      this.errorMessage = '';
      this.navCtrl.navigateForward('/home');
    }, err => {
      this.errorMessage = err.message;
    });
}
```

Notou que tiramos o texto da atribuição a `this.errorMessage` e que inserimos a instrução `this.navCtrl.navigateForward('/home');`? Mas de onde vem este `navCtrl`? Precisamos injetá-lo em nosso construtor com o código a seguir.

```
private navCtrl: NavController
```

Para testarmos nossa aplicação, vamos criar uma página básica para Peças com a instrução a seguir no terminal. No template, você pode colocar um texto qualquer dentro de `<ion-content>`, apenas para ver a execução da aplicação.

```
ionic generate page pecas
```

Mas isso ainda não é o suficiente. Lembra que falei sobre o elemento `tab` dentro do `<ion-tab-button>`? Pois bem, a aplicação buscará por

uma rota chamada `pecas` e não a temos. Se você lembrar da implementação do Sidemenu, tivemos que implementar um sistema de rota específico para ele. Precisaremos fazer o mesmo aqui, pois a navegação é outra após a autenticação. Não pode mais levar em consideração as rotas do `app-routing.module.ts`. Precisamos então no arquivo `home-routing.module.ts` da pasta `home`, implementar o código a seguir. Retire qualquer referência à página `home` que tenha na definição de `routes`.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomePage } from './home.page';
import { PecasPage } from '../pecas/pecas.page';

const routes: Routes = [
  {
    path: 'tabsMenu',
    component: HomePage,
    children: [
      {
        path: 'pecas',
        children: [
          {
            path: '',
            loadChildren: () => import('../pecas/pecas.module').then(m =>
m.PecasPageModule)
          }
        ]
      },
    ]
  },
  {
    path: '',
    redirectTo: 'tabsMenu/pecas',
    pathMatch: 'full'
  }
  ];

@NgModule({
```

```
  imports:
    [
      RouterModule.forChild(routes)
    ],
  exports:
    [
      RouterModule
    ]
})
export class HomePageRoutingModule {}
```

Veja a figura a seguir que apresenta a página com com os `Tabs` e e página de Peças renderizada.

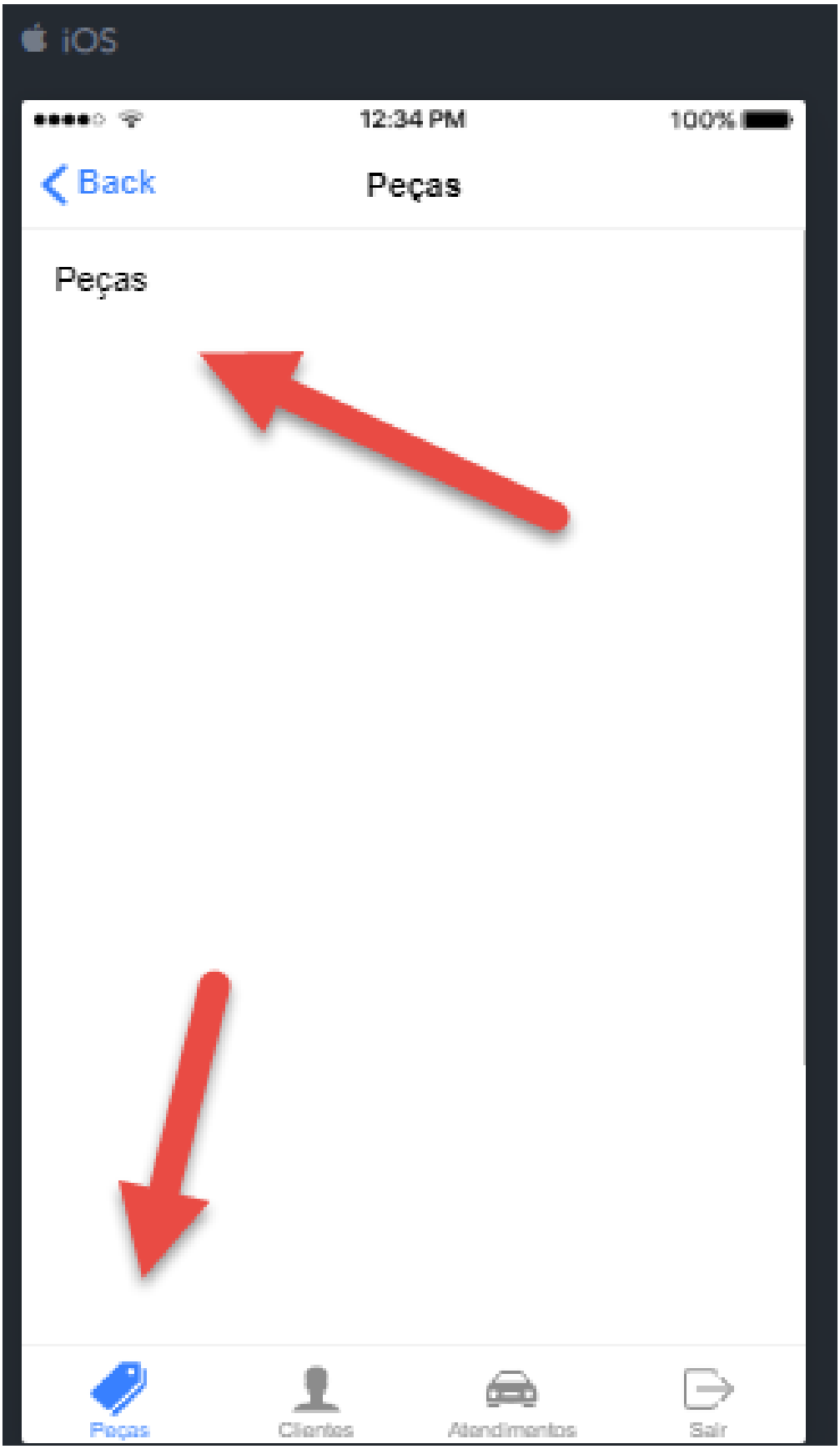

iOS
12:34 PM
100%
Back
Peças
Peças
Peças
Clientes
Atendimentos
Sair

Figura 9.3: Página com Tabs e a de Peças renderizada

9.4 A página para listagem de clientes registrados

Para testarmos o acesso ao conteúdo do Firebase apenas quando nosso usuário estiver conectado, vamos trazer para este novo projeto nosso componente de listagem de clientes, que implementamos no capítulo anterior. Pode copiar a pasta `listagem` dentro de `clientes` para uma pasta chamada `clientes` dentro do nosso projeto deste capítulo. Para auxiliar, trago na sequência as implementações destes arquivos. Foram mantidas apenas as instruções relativas à listagem.

clientes-listagem.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

import { IonicModule } from '@ionic/angular';

import { ClientesListagemPageRoutingModule } from './clientes-listagem-routing.module';

import { ClientesListagemPage } from './clientes-listagem.page';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    ClientesListagemPageRoutingModule
  ],
  declarations: [ClientesListagemPage]
```

```
})
export class ClientesListagemPageModule {}
```

clientes-listagem.page.html

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
    </ion-buttons>
    <ion-title>Clientes</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content padding-left padding-top>
  <ion-list #slidingList>
    <ion-item-sliding *ngFor="let cliente of clientes">
      <ion-item routerDirection="forward" [routerLink]="['/', 'clientes-
add-edit', cliente.clienteid]">
        <ion-thumbnail slot="start">
          <ion-img [src]="cliente.foto"></ion-img>
        </ion-thumbnail>
        <ion-grid>
          <ion-row>
            <ion-label>
              <p style="font-size: larger; font-weight: bold;">
{{cliente.nome}}</p>
            </ion-label>
          </ion-row>
          <ion-row>
            <ion-col col-12>
              <ion-label>
                <p style="font-weight: 400; text-align: right">Telefone:
{{cliente.telefone}}</p>
              </ion-label>
            </ion-col>
          </ion-row>
        </ion-grid>
      </ion-item>

      <ion-item-options side="end">
```

```
        <ion-item-option color="danger" (click)="removerCliente(cliente)">
          <ion-button color="danger" no-padding>
            <ion-icon name="trash"></ion-icon>
            Remover
          </ion-button>
        </ion-item-option>
      </ion-item-options>
    </ion-item-sliding>
  </ion-list>

  <ion-fab vertical="top" horizontal="end" slot="fixed" edge="true">
    <ion-fab-button routerDirection="forward" [routerLink]="['/clientes-
add-edit/-1']">
      <ion-icon name="add"></ion-icon>
    </ion-fab-button>
  </ion-fab>
</ion-content>
```

clientes-listagem.page.ts

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { Capacitor } from '@capacitor/core';
import { IonList } from '@ionic/angular';
import { Cliente } from 'src/app/models/cliente.model';
import { ClientesService } from 'src/app/services/clientes.service';

@Component({
  selector: 'app-clientes-listagem',
  templateUrl: './clientes-listagem.page.html',
  styleUrls: ['./clientes-listagem.page.scss'],
})
export class ClientesListagemPage implements OnInit {

  public clientes: Cliente[];
  @ViewChild('slidingList') slidingList: IonList;

  constructor(
    private clientesService: ClientesService,
  ) { }
```

```
  ngOnInit() {
  }

  async ionViewWillEnter() {
    this.clientes = await this.clientesService.getAll();
  }

  caminhoFotoParaListagem(foto: string) {
    if (foto != '') {
      return Capacitor.convertFileSrc(foto);
    } else {
      return 'assets/imgs/icon_clientes.png';
    }
  }
}
```

Sabemos que junto desta página temos outros componentes ligados e precisamos também trazê-los para nossa aplicação. São eles:

- `clientes.service.ts` , que está na pasta `services` e deverá ser trazido para a pasta de mesmo nome;
- `cliente.model.ts` , que está na pasta `models` e deverá ser trazido para uma pasta de mesmo nome, a ser criada na aplicação deste capítulo.

Para facilitar, trago na sequência estes códigos também, apenas com o que será pertinente para este momento.

clientes.service.ts

```
import { Injectable } from '@angular/core';
import { Cliente, clienteConverter } from '../models/cliente.model';
import { Firestore, collection, getDocs, query, orderBy, } from
'@angular/fire/firestore';


@Injectable({
    providedIn: 'root'
})
export class ClientesService {
```

```
    constructor(
        private _fireStore: Firestore,
    ) {
    }

    async getAll(): Promise<Cliente[]> {
        const clientes: Cliente[] = [];
        const q = query(collection(this._fireStore, "clientes"),
orderBy("nome")).withConverter(clienteConverter);
        const querySnapshot = await getDocs(q);
        querySnapshot.forEach((doc) => {
            clientes.push(doc.data());
        });
        return clientes;
    }
}
```

cliente.model.ts

```
export interface Cliente {
    clienteid: string;
    nome: string;
    email: string;
    telefone: string;
    renda: number;
    nascimento: Date;
    foto: string;
    nomeDaFoto: string;
}


export const clienteConverter = {
    toFirestore: (cliente) => {
        return <Cliente>{
            nome: cliente.nome,
            email: cliente.email,
            telefone: cliente.telefone,
            renda: cliente.renda,
            nascimento: cliente.nascimento,
            foto: cliente.foto,
```

```
            nomeDaFoto: cliente.nomeDaFoto,
        };
    },
    fromFirestore: (snapshot, options) => {
        const data = snapshot.data(options);
        return <Cliente>{
            clienteid: snapshot.id,
            nome: data.nome,
            email: data.email,
            telefone: data.telefone,
            renda: data.renda,
            nascimento: data.nascimento.toDate(),
            foto: data.foto,
            nomeDaFoto: data.nomeDaFoto,
        }
    }
};
```

Como agora nossa aplicação faz uso do Firestore, precisamos importar em nosso `app.module.ts` os componentes necessários. Sendo assim, insira as instruções a seguir ao final de `imports` do `@NgModule()`. Imports serão necessários.

```
provideFirebaseApp(() => {
  return initializeApp(environment.firebaseConfig);
}),
provideFirestore(() => {
  if (environment.useEmulators) {
    const firestore = getFirestore();
    enableIndexedDbPersistence(firestore);

    return firestore;
  } else {
    getFirestore();
  }
}),
```

Tal qual fizemos no capítulo 8, precisaremos importar o `rxfire` e o faremos no terminal, com a instrução a seguir.

```
npm install rxfire
```

Para podermos testar, precisaremos inserir a rota a seguir no mesmo nível de `peças` em `home-routing.module.ts`. Fique atento à estrutura onde está o arquivo.

```
{
    path: 'clientes',
    children: [
      {
        path: '',
        loadChildren: () => import('../clientes-listagem/clientes-
listagem.module').then(m => m.ClientesListagemPageModule)
      }
    ]
},
```

Caso no capítulo anterior tenha sido gravado algum cliente sem foto, precisamos trazer para nosso projeto a figura que utilizamos como padrão para o caso de ausência de uma foto. Sendo assim, traga a figura `icon_clientes.png` da pasta `assets/imgs` para nossa aplicação deste capítulo, para o mesmo endereço e com o mesmo nome.

Muito bem, agora já podemos testar nossa aplicação. Veja na figura a seguir a representação obtida para esta implementação.

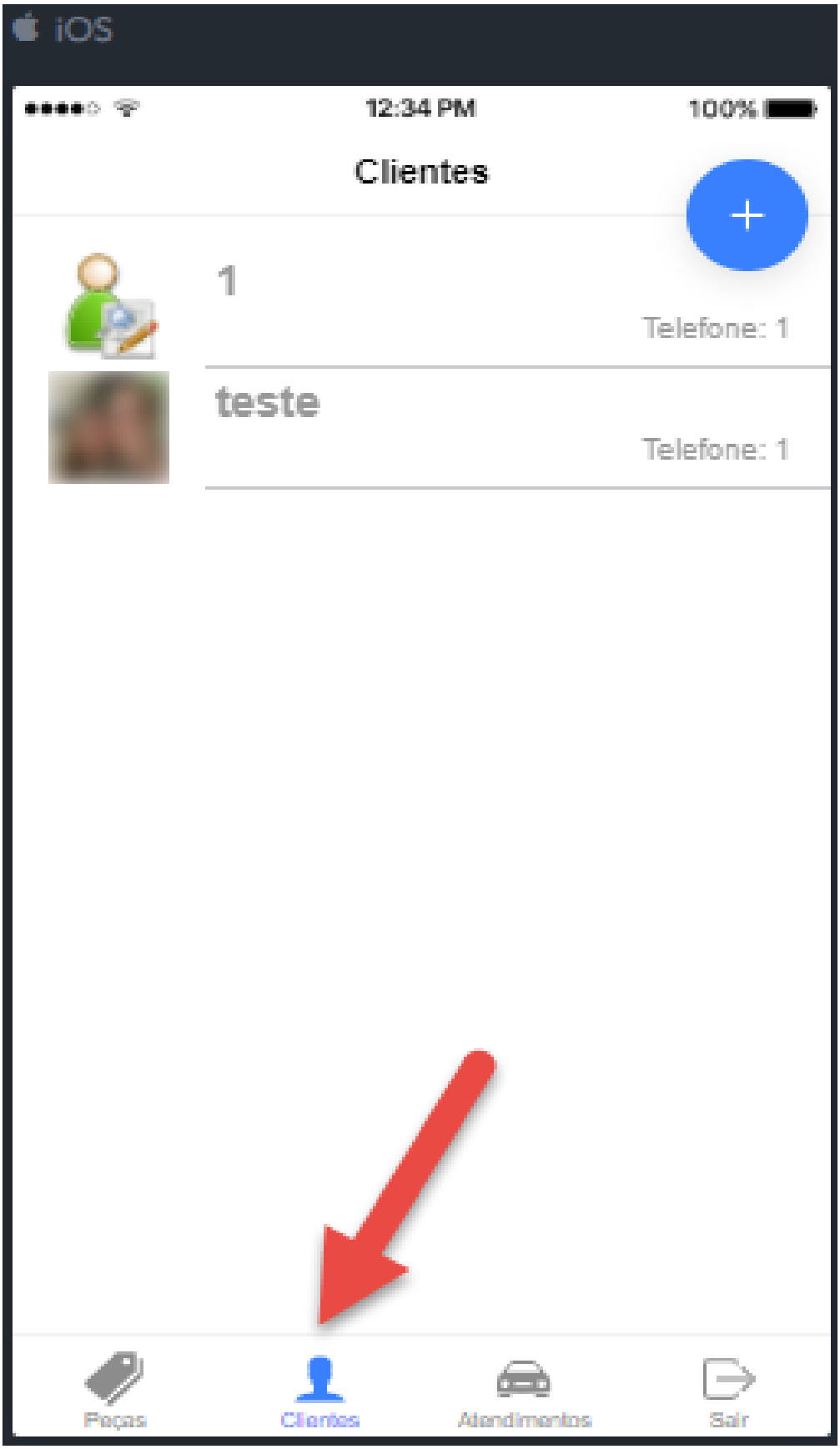

iOS
12:34 PM
100%
Clientes
1
Telefone: 1
teste
Telefone: 1
Peças
Clientes
Atendimentos
Sair

Figura 9.4: Página com Tabs e a de Clientes renderizada

9.5 Proibindo o acesso anônimo

Com a configuração que temos para acesso em nossa aplicação no Firebase, quer seja o Firestore ou o Storage, qualquer usuário com as credenciais acessa e não é isso que queremos. Precisamos corrigir isso, atribuindo uma regra mínima de autenticação para o acesso. Vamos então acessar o menu `Firestore Database` e depois `Regras` e vamos substituir a regra que lá está pela informada na sequência. A atualização e disponibilização para sua aplicação pode demorar alguns momentos.

```
allow read, write: if request.auth.uid != null;
```

Vamos realizar o mesmo procedimento para o `Storage`, inserindo em regras o mesmo código apresentado anteriormente. Para testarmos, vamos alterar nossa rota padrão, que está no `app-routing.module.ts` para não forçar a tela de login de início. Comente a rota `home` e em `clientes-listagem`, deixe o `path` vazio, igual ao `home` comentado. Ative a janela de desenvolvedor no navegador (pressionando F12) e veja na console o erro apresentado na figura a seguir.

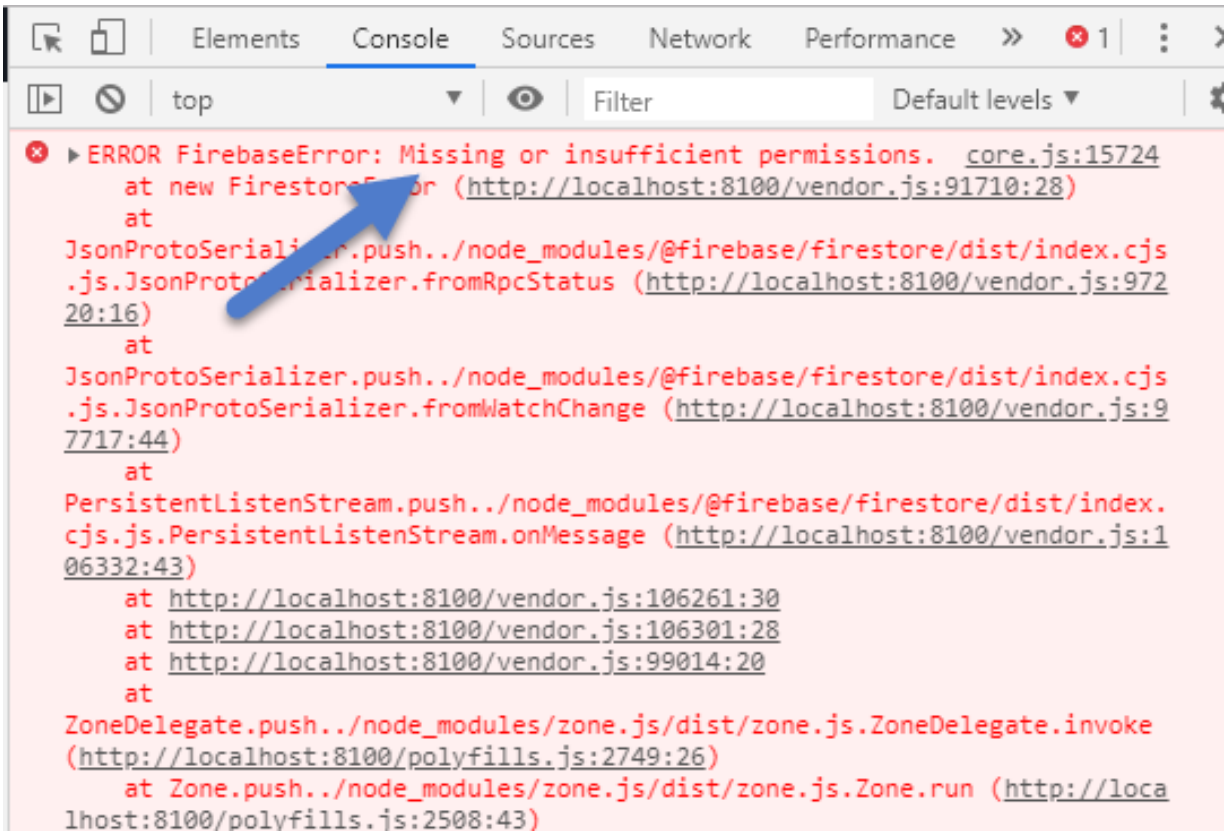


Figura 9.5: Erro ao acessar o Firebase sem autenticação

Nossos dados estão protegidos agora, podendo ser acessados apenas para quem se autenticar. Precisamos então adaptar nossa aplicação para que leve isso em consideração. Vamos começar estas alterações na classe `AutenticacaoService`, inserindo nela o método a seguir, que nos retornará um valor booleano informando se temos ou não um usuário autenticado.

```
public isUserAuthenticated(): boolean {
  return ((getAuth().currentUser !== null));
}
```

Na sequência, precisamos alterar nosso método `getAll()` the `ClientesService`, para que, antes de realizar a leitura dos dados, verifique se nosso usuário está autenticado. Como consumiremos o novo método que implementamos pela listagem anterior, vamos realizar a injeção de `AutenticacaoService` em nosso construtor de `ClientesService`, com a instrução a seguir.

```
private autenticacaoService: AutenticacaoService
```

Agora sim, vamos consumir o novo método. Veja a nova implementação para `getAll()` na sequência. Observe que preferimos trabalhar com exceções caso não haja autenticação no Firebase.

```
async getAll(): Promise<Cliente[]> {
        if (!this.autenticacaoService.isUserAuthenticated()) {
            throw new Error('Usuário não está autenticado');
        }
        const clientes: Cliente[] = [];
        const q = query(collection(this._fireStore, "clientes"),
orderBy("nome")).withConverter(clienteConverter);
        const querySnapshot = await getDocs(q);
        querySnapshot.forEach((doc) => {
            clientes.push(doc.data());
        });
        return clientes;
}
```

E como fica nossa classe que renderiza o template visual? É nela que temos a invocação do `getAll()` que retornará os clientes a serem exibidos. Vamos trabalhar na situação em que, caso uma exceção seja disparada, um alerta será exibido ao usuário. Caso não haja exceção, os dados serão atribuídos e então exibidos.

Vamos utilizar nossa classe `AlertService`, que já temos implementada de capítulos anteriores. Para facilitar, trago o código necessário, que deve ser implementado no arquivo `alert.service.ts`, logo na sequência.

```
import { Injectable } from '@angular/core';
import { AlertController } from '@ionic/angular';

@Injectable({
    providedIn: 'root'
  })
export class AlertService {
    constructor(private alertCtrl: AlertController) {}
```

```
    async presentAlert(header: string, subHeader: string, message: string,
buttons: string[]) {
        const alert = await this.alertCtrl.create({
            header,
            subHeader,
            message,
            buttons
        });
        await alert.present();
    }
}
```

Por fim, vamos adaptar nosso método `ionViewWillEnter()` da listagem de clientes para esta nova situação. Veja-o na listagem a seguir.

```
ionViewWillEnter() {
    try {
        this.clientes = await this.clientesService.getAll();
    } catch (error) {
        this.alertService.presentAlert('Erro de conexão', 'Usuário não
está autenticado', 'Tente conectar novamente', [ 'OK'] );
    }
}
```

Lembre-se de que precisamos injetar `alertService` em nosso construtor, com a instrução a seguir.

```
private alertService: AlertService
```

Agora podemos testar nossa aplicação. Veja que o alerta é exibido, pois não temos autenticação de usuário. Retorne as rotas do `app-routing.module.ts` ao normal, tendo `login` como acesso inicial e teste o acesso autenticado.

9.6 Criação de novos usuários diretamente

pela aplicação

É muito comum ao acessarmos sites ou aplicativos em nossos dispositivos e eles nos solicitarem autenticação e, em conjunto, a possibilidade de criarmos uma nova conta para este acesso. O Firebase Authentication nos oferece de maneira bem simples este recurso e é o que implementaremos nessa seção. Vamos começar pelo método em nossa classe `AutenticacaoService`, que está na sequência.

```
async registrar(value) {
    initializeApp(environment.firebaseConfig);
    const auth = getAuth();

    try {
      return await createUserWithEmailAndPassword(auth, value.email,
value.password);
    } catch (error) {
      throw new Error(error.message);
    }
}
```

Notou o quão facilitadora é a API oferecida pelo Firebase? Apenas invocamos um método, enviando a ele o email e senha para a conta que precisará ser criada.

Agora precisamos criar nossos arquivos necessários para esta nova página, que se chamará `registrar`. Vamos executar a instrução a seguir? Com a aplicação em execução, elimine, se quiser, o arquivo de teste, como estamos fazendo desde o início. A instrução criará nossos arquivos dentro de uma pasta chamada `usuarios`.

```
ionic generate page usuarios/registrar
```

Trabalharemos alguns estilos em nosso template, vamos então já codificá-los em nosso arquivo `registrar.page.scss`:

```
.validation-errors {
    text-align: center;
```

```
    color: red;;
}

.validation-success {
    text-align: center;
    color: green;;
}
```

Podemos agora implementar nosso template.

```
<ion-header>
  <ion-toolbar color="primary">
    <ion-title>Registrar usuário</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content padding>
  <form class="form" [formGroup]="validationsForm"
(ngSubmit)="registrar(validationsForm.value)">

    <ion-item>
      <ion-label  position="floating" color="primary">Email</ion-label>
      <ion-input type="text" formControlName="email"></ion-input>
    </ion-item>
    <div class="validation-errors">
      <ng-container *ngFor="let validation of validationMessages.email">
        <div class="error-message"
*ngIf="validationsForm.get('email').hasError(validation.type) &&
(validationsForm.get('email').dirty ||
validationsForm.get('email').touched)">
          {{ validation.message }}
        </div>
      </ng-container>
    </div>

    <ion-item>
      <ion-label  position="floating" color="primary">Senha</ion-label>
      <ion-input type="password" formControlName="password" class="form-
controll" required></ion-input>
    </ion-item>
```

```
    <div class="validation-errors">
      <ng-container *ngFor="let validation of
validationMessages.password">
        <div class="error-message"
*ngIf="validationsForm.get('password').hasError(validation.type) &&
(validationsForm.get('password').dirty ||
validationsForm.get('password').touched)">
          {{ validation.message }}
        </div>
      </ng-container>
    </div>

    <div style="text-align: center">
      <ion-button  class="submit-btn" type="submit"
[disabled]="!validationsForm.valid">Registrar</ion-button>
      <br/><label class="validation-errors">{{errorMessage}}</label>
      <br/><label class="validation-success">{{successMessage}}</label>
    </div>
  </form>
  <div style="text-align: center">
    <p class="go-to-login">Já tem uma conta? <a
(click)="goLoginPage()">Tente se autenticar</a></p>
  </div>
</ion-content>
```

Para renderizarmos nosso template, precisamos implementar nossa classe em TypeScript e podemos ver o código dela na sequência.

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormBuilder, Validators, FormControl } from
'@angular/forms';
import { AutenticacaoService } from '../../services/autenticacao.service';
import { NavController } from '@ionic/angular';

@Component({
  selector: 'app-register',
  templateUrl: './registrar.page.html',
  styleUrls: ['./registrar.page.scss'],
})
export class RegistrarPage implements OnInit {
```

```
  validationsForm: FormGroup;
  errorMessage: '';
  successMessage: string;

  validationMessages = {
   email: [
    { type: 'required', message: 'Informe o e-Mail' },
    { type: 'pattern', message: 'Informe um email válido' }
   ],
   password: [
    { type: 'required', message: 'Informe a senha' },
    { type: 'minlength', message: 'A senha precisa ter ao menos 6
caracteres' }
  ]
 };

  constructor(
    private navCtrl: NavController,
    private autenticacaoService: AutenticacaoService,
    private formBuilder: FormBuilder
  ) {}

  ngOnInit() {
    this.validationsForm = this.formBuilder.group({
      email: new FormControl('', Validators.compose([
        Validators.required,
        Validators.pattern('^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+.[a-zA-Z0-
9-.]+$')
      ])),
      password: new FormControl('', Validators.compose([
        Validators.minLength(6),
        Validators.required
      ])),
    });
  }

  registrar(value) {
    this.autenticacaoService.registrar(value)
```

```
      .then(res => {
        this.errorMessage = '';
        this.successMessage = 'Sua conta foi criada com sucesso. Se
autentique.';
      }, err => {
        this.errorMessage = err.message;
        this.successMessage = '';
      });
  }

  goLoginPage() {
    this.navCtrl.navigateBack('');
  }
}
```

Para finalizar esta implementação, precisamos importar `ReactiveFormsModule` no arquivo `registrar.module.ts`, pois estamos utilizando o `Reactive Forms` na página.

Quando criamos nossa página de registrar usuário, uma nova rota foi inserida em nosso `app-routing-module.ts`, direcionando para nossa nova página. Em nossa página de login, precisamos dar ao usuário a possibilidade de navegar para esta página. Vamos alterar o link que já temos lá, para o código a seguir.

```
<p style="text-align: center">
    Não tem conta ainda? <a (click)="goToRegistrarPage()">Crie uma
conta</a>
</p>
```

Notou a ligação do `click` com o método `goToRegistrarPage()`? Então já sabe que precisamos implementá-lo em nossa classe `LoginPage`, tal qual podemos ver na sequência.

```
goToRegistrarPage() {
    this.navCtrl.navigateForward('/registrar');
}
```

Teste a aplicação e vá para a criação de usuários. Insira um email e senha válidos e registre. Veja o retorno na página. Vá para o

Firebase e veja os usuários registrados. Depois tente acessar a aplicação com o novo usuário. Na sequência, veja uma figura apresentando a página que acabamos de criar.

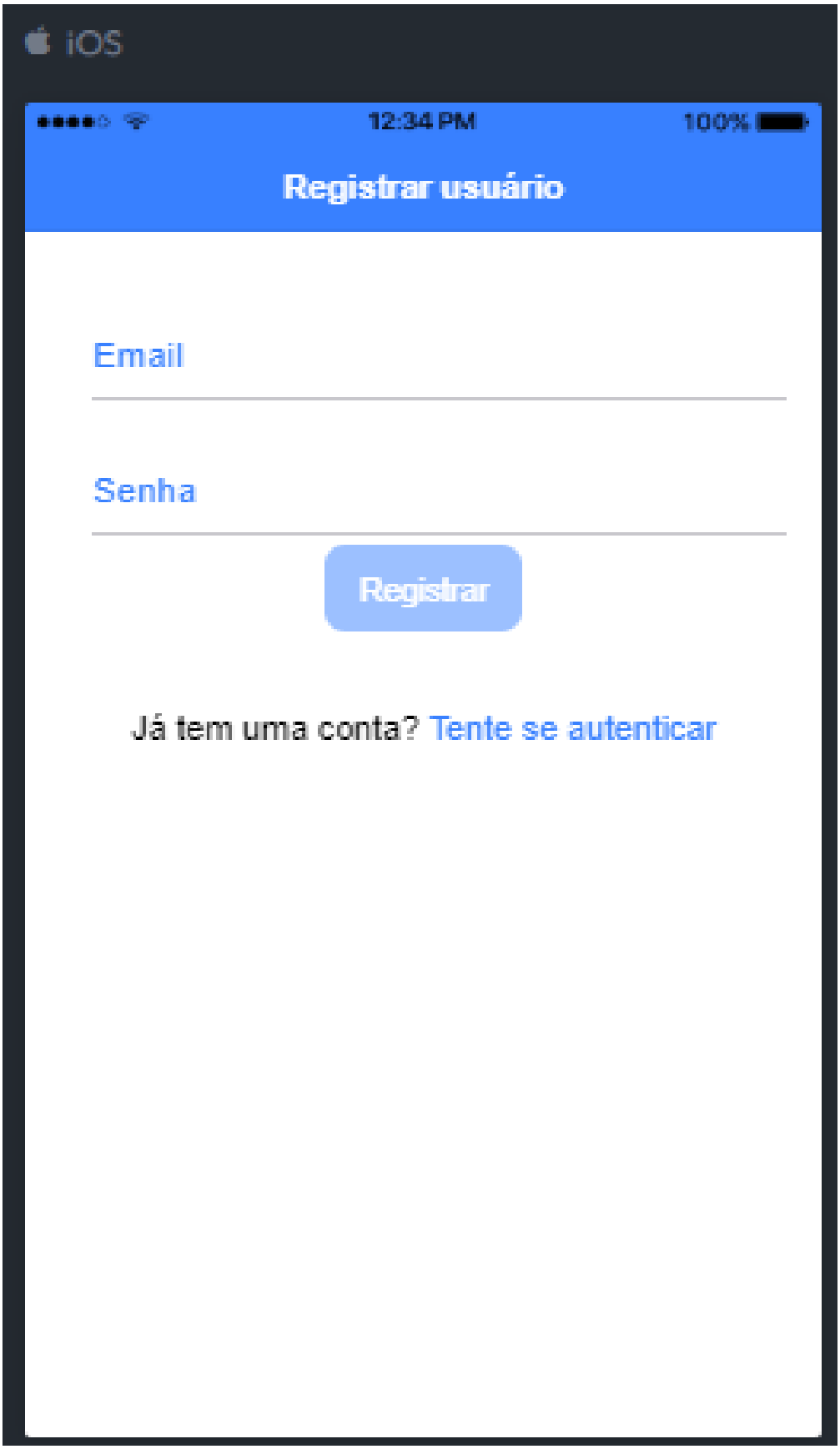

iOS
12:34 PM
100%
Registrar usuário
Email
Senha
Registrar
Já tem uma conta? Tente se autenticar

Figura 9.6: Registro de um novo usuário no Firebase

Conclusão

Fechamos mais um capítulo. Trabalhamos a autenticação de um usuário em nossa aplicação por meio do Firebase Authentication. Possibilitamos o uso de regras que não permitem acesso sem que uma autenticação tenha sido realizada pela aplicação. Utilizamos a API do Firebase também para criarmos um usuário e já o deixarmos apto a acessar nossa aplicação.

Na parte específica do Ionic vimos o `<ion-tabs>`, uma página que possibilita a navegação entre outras por meio de botões na base de uma página principal.

Em relação ao `Authentication`, o Firebase possibilita que configuremos nossa aplicação para que o usuário receba um email após a criação de seu acesso e que o ative apenas por meio deste email. Também está disponível login por outras plataformas, como Google e Facebook, dentre várias outras disponíveis no Console do Firebase. Você pode pensar em trazer isso para sua aplicação. É algo muito comum nos dias atuais e evita que o usuário fique criando contas para todos os aplicativos.

Estamos próximos do término do livro. Teremos só o próximo capítulo para vermos. Trabalharemos o uso de mapas. Será um fechamento bem legal para nosso trabalho.

Capítulo 10
Integração com Google Maps

Neste último capítulo de conteúdo do nosso livro, trabalharemos um pouco com mapas, o que se tornou algo comum em aplicações web e para dispositivos móveis e não poderíamos deixar de ver isso aqui. Nós faremos uso da API do Google Maps, com auxílio dos plugins `@capacitor-community/capacitor-googlemaps-native` e `@capacitor/geolocation`. Veremos os passos para instalação e testes. Executaremos nossa aplicação em um dispositivo Android, mas o procedimento para o iOS será relatado no que for diferente. O funcionamento em emuladores para esta funcionalidade não é muito bom e em alguns casos nem chega a funcionar.

Preparando o ambiente para o uso de mapas

O primeiro passo para esta última aplicação será criarmos um novo projeto. Você pode utilizar os projetos anteriores, mas preferi aqui criar um novo. Em seu terminal, na pasta inicial de seus projetos, digite a instrução a seguir.

```
ionic start capitulo10 blank --type=angular --no-git
```

Com nosso projeto criado, começaremos a instalação de plugins que serão necessários para nossa aplicação. Execute a instrução a seguir, no terminal, na pasta do projeto criado anteriormente.

```
npm install --save @capacitor-community/capacitor-googlemaps-native
@ionic-native/core @capacitor/geolocation
```

10.1 Criando nossa aplicação no Google Cloud

Para que possamos fazer uso de qualquer API do Google, como o caso do Google Maps, precisamos realizar um registro no serviço de Cloud do Google. Para isso, acesse o link https://cloud.google.com/maps-platform/#get-started e você terá acesso a uma página com um formulário semelhante ao que se apresenta na figura a seguir. Marque `Maps` e `Places` nele. Tudo isso se você já tiver a conta criada e configurada. Caso não, continue lendo. Talvez as interfaces referentes ao portal do Google e Google maps apresentadas se alterem.

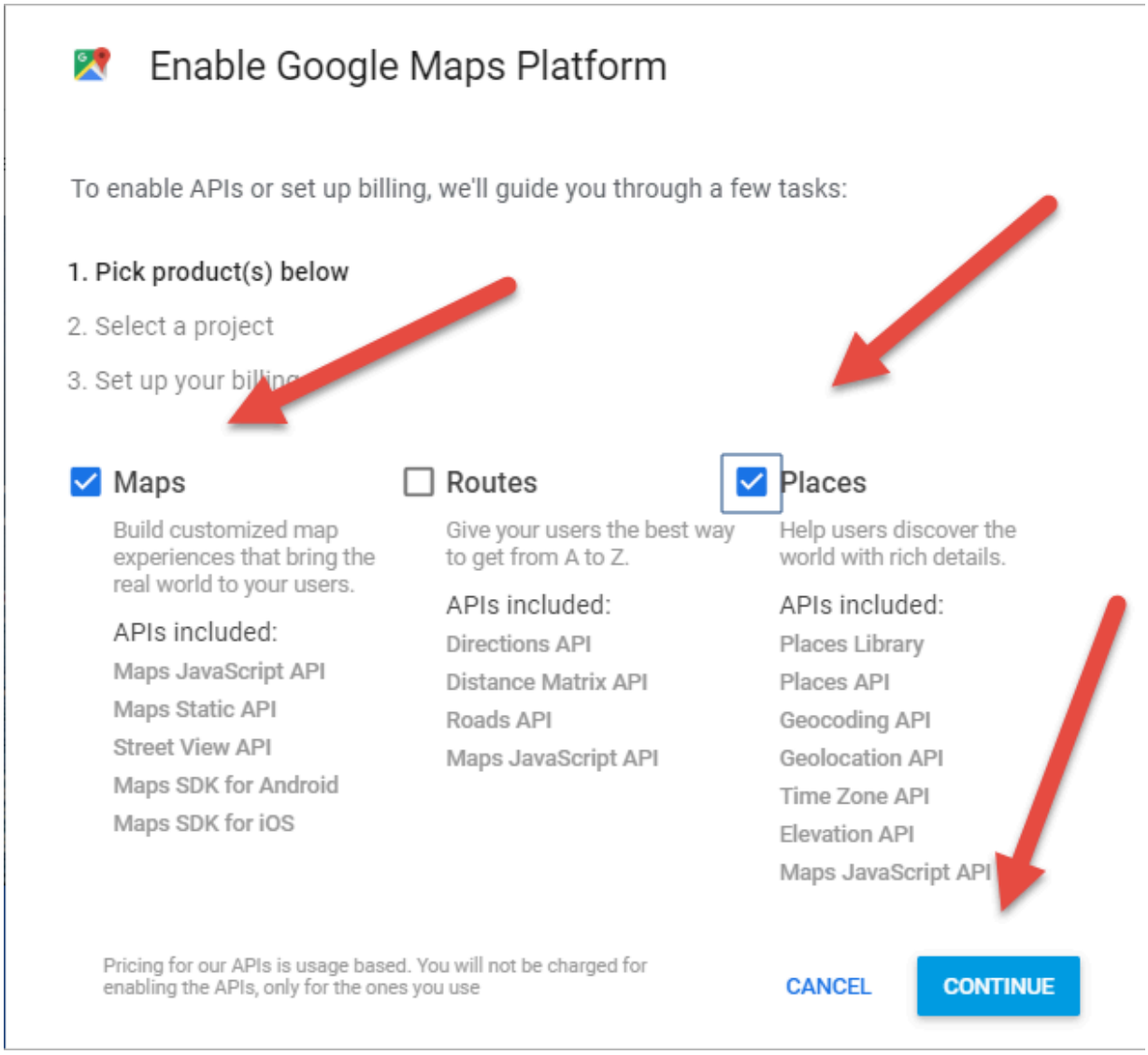


Figura 10.1: Selecionando a API de mapas do Google

Após esta seleção, uma nova etapa para o formulário será exibida, agora pedindo para selecionarmos um projeto ou criarmos um novo. Podemos optar pela criação de um novo, caso você ainda não o tenha. Veja a figura a seguir como exemplo.

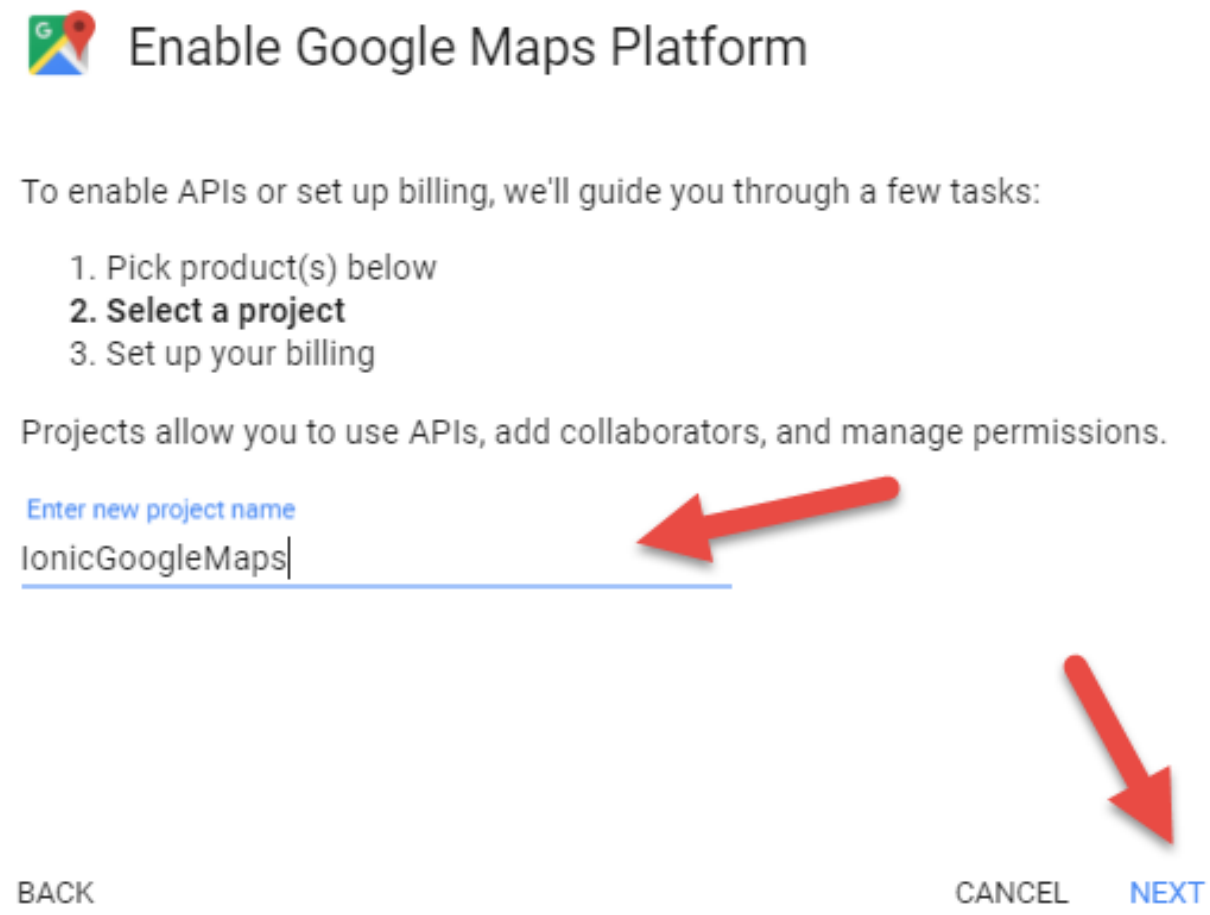


Figura 10.2: Criando/selecionando a aplicação Google

Até um tempo atrás, o Google não solicitava um cartão de crédito para liberação do uso de algumas APIs. Isso mudou. Precisamos informar um cartão, mas nada será cobrado de imediato, mas se houver um uso maior que a franquia apresentada, a cobrança poderá ocorrer. Isso é informado após a confirmação da etapa anterior, com o surgimento da solicitação de confirmação apresentada na figura a seguir.

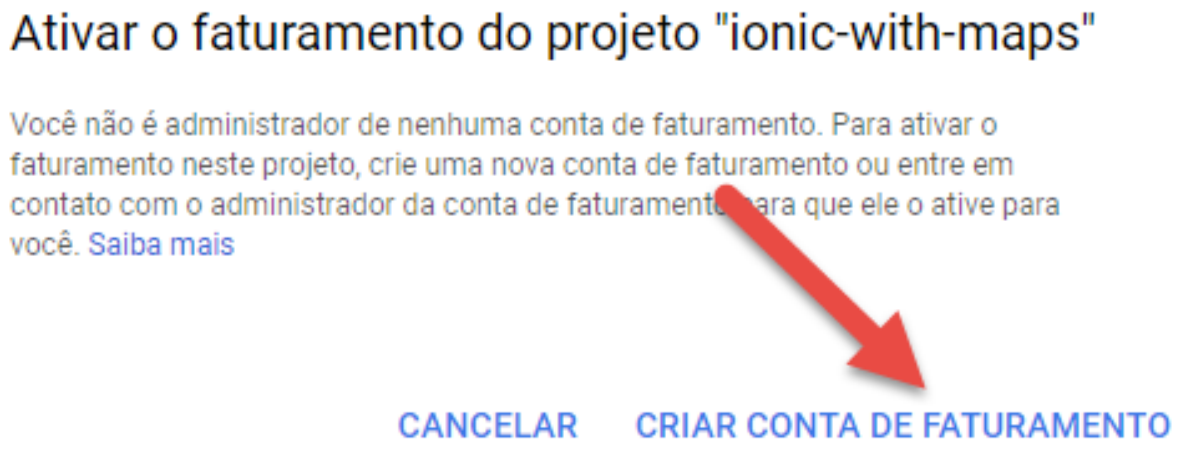


Figura 10.3: Confirmando aceite para informação de cartão

A próxima etapa é a informação de seus dados de contato e cartão de crédito.

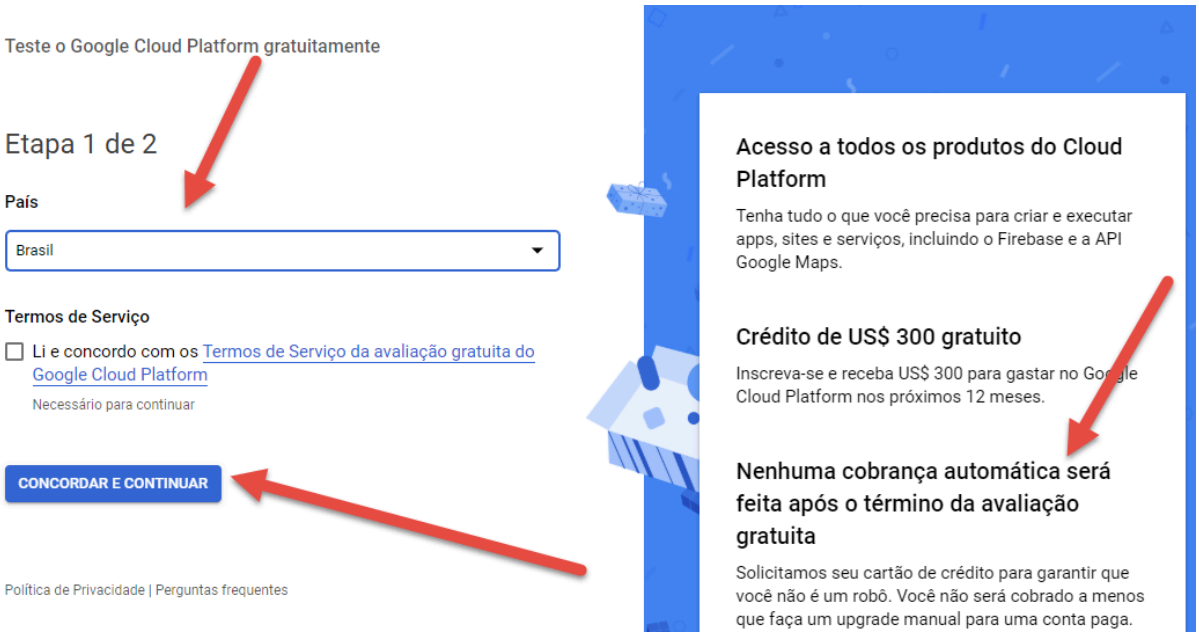


Figura 10.4: Informação de dados de contato e cartão

Após confirmar todos seus dados, uma última etapa será visualizada, solicitando confirmação para ativação da plataforma do Google Maps. Verifique todos os exemplos sugeridos até a finalização da etapa.

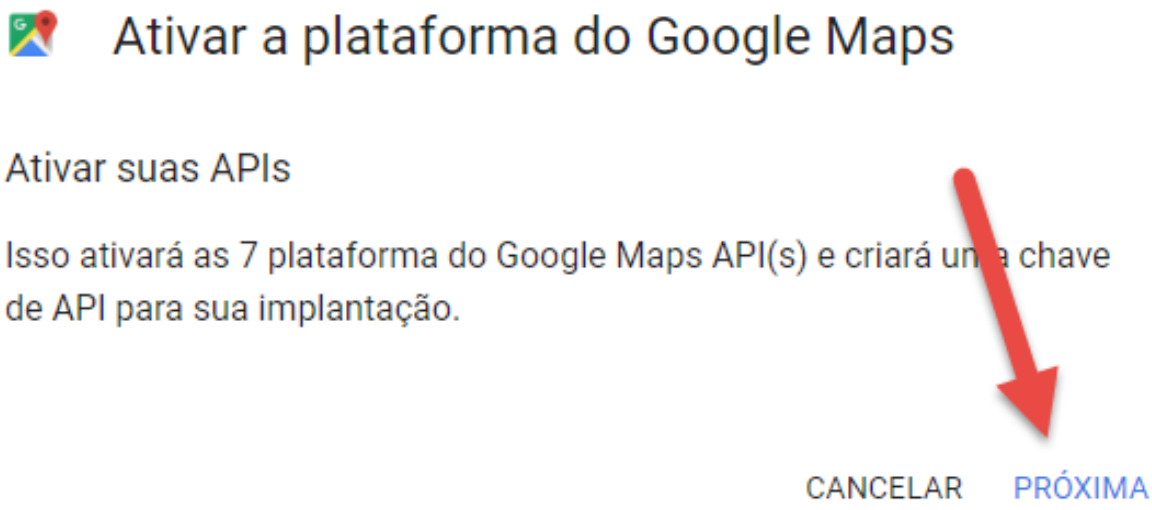


Figura 10.5: Ativação da plataforma Google Maps

Para finalizar, será solicitada a você a criação de credenciais para sua aplicação. Após criá-la, copie, pois a utilizaremos em nossas implementações, como já veremos. Você precisará instalar a API para Android e iOS, assim como criar as credenciais respectivas.

10.2 Preparação da aplicação para a visualização de mapas

Nossa aplicação terá uma página de abertura, a `home`, que é criada automaticamente quando criamos nosso projeto. Colocaremos nela uma imagem estática, de apresentação da aplicação e teremos o acesso ao mapa por meio de opções do `Hamburger Menu`.

Já temos então todos os artefatos para a página `home` criados. Vamos agora implementar o que precisamos para essa proposta. Os códigos estão na sequência, em subseções.

home.page.html

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
      <ion-menu-button></ion-menu-button>
    </ion-buttons>
    <ion-title>
      Aplicação Google Maps
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content>
  <div class="ion-padding">
    <ion-avatar class="main" style="height: 290px;">
      <ion-img src="assets/images/googlemaps.jpg" class="center"></ion-
img>
    </ion-avatar>
  </div>
  <div>
    <ion-card>
      <ion-card-header>
        <ion-card-subtitle>Ionic 5</ion-card-subtitle>
        <ion-card-title>Google Maps Plugin</ion-card-title>
      </ion-card-header>

      <ion-card-content>
        Este aplicativo trará exemplos de utilização da API do Google Maps
por meio do
        <b>capacitor-community/capacitor-googlemaps-native</b>
        e <b>@ionic-native/google-map</b>. Acesse as opções pelo
<b>Hamburguer menu</b>
      </ion-card-content>
    </ion-card>
  </div>
</ion-content>
```

Veja os componentes do `<ion-header>` no código anterior. Temos o botão para o `Hamburger Menu` e o título da página. Já no `<ion-content>` temos dois `<div>`. O primeiro para a figura, que devemos ter no

caminho especificado. Esta figura está dentro de um `<ion-avatar>`, em um `<ion-img>`, e você precisa trazê-la para seu projeto. O `<ion-avatar>`, segundo `<div>`, é um componente que formata uma imagem por meio do `<ion-card>`, para ser exibida como um círculo, forma usual para algumas aplicações exibirem imagens. A `ion-padding` é oferecida pelo Ionic.

home.page.scss

```
.main {
    width : 100%;
}

.center {
    margin : 0 auto;
}

ion-avatar ion-img {
    width: 300px !important;
    height: 300px !important;
}
```

As classes de estilo que criamos pelo código anterior, `main` e `center`, permitirão a centralização da imagem na página, já a customização do `ion-img`, dentro do `ion-avatar`, define o tamanho da imagem.

Não precisamos implementar nada no código que renderizará nosso template, mas e o menu que queremos exibir para o usuário acessar o mapa? Vamos implementá-lo em nosso componente `app`. Também não mexeremos no `home.module.ts`. Vamos então ver os códigos necessários.

app.component.html

```
<ion-split-pane contentId="main-content">
  <ion-menu contentId="main-content" type="overlay">
    <ion-header>
      <ion-toolbar>
```

```
        <ion-title>Menu</ion-title>
      </ion-toolbar>
    </ion-header>
    <ion-content>
      <ion-list>
        <ion-menu-toggle auto-hide="false" *ngFor="let p of paginasApp">
          <ion-item [routerDirection]="'root'" [routerLink]="[p.url]">
            <ion-icon slot="start" [name]="p.icon"></ion-icon>
            <ion-label>
              {{p.titulo}}
            </ion-label>
          </ion-item>
        </ion-menu-toggle>
      </ion-list>
    </ion-content>
  </ion-menu>
  <ion-router-outlet id="main-content"></ion-router-outlet>
</ion-split-pane>
```

Temos os componentes `<ion-split-pane>` e o `<ion-menu-toggle>`, que vimos quando criamos o `Sidemenu` no capítulo 6. Temos um `<ion-item>` com informações ligadas à nossa classe, que veremos daqui a pouco. Veja que este componente está dentro de um `<ion-list>` e ele tem, em sua definição, o `*ngFor`. Ao final, antes do fechamento de `<ion-split-pane>`, temos `<ion-router-outlet main></ion-router-outlet>`, que é um componente que integra as rotas com o Angular.

app.component.ts

```
import { Component } from '@angular/core';


@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html'
})
export class AppComponent {
  public paginasApp = [
    {
```

```
      titulo: 'Home',
      url: '/home',
      icon: 'home'
    },
    {
      titulo: 'Minha localização',
      url: '/mylocation',
      icon: 'pin'
    }
  ];

  constructor() {}
}
```

No código anterior, referente ao `app.component.ts`, temos a declaração do objeto `paginasApp`. Temos um título, uma URL e um ícone. Esses dados comporão as opções do menu que será exibido pelo HTML que vimos há pouco. Temos dois objetos criados, mas no momento temos apenas a página `home`. A `mylocation` veremos mais adiante. Não mexeremos em `app-routing.module.ts` e nem em `app.module.ts`, pois manteremos o código gerado na criação da aplicação.

10.3 A aplicação em execução

O `<ion-split-pane>` é um componente que possibilita que mais de uma página seja visualizada ao mesmo tempo - já comentamos que isso é comum em tablets.

Ainda que não estejamos utilizando de maneira explícita os componentes e plugins para o Google Maps, precisamos registrar em nossa aplicação a API KEY que geramos anteriormente e darmos algumas permissões para a aplicação. Para o Android, isso é feito no arquivo `\android\app\strc\main\AndroidManifest.xml`. Veja o código anterior e as instruções de onde inseri-lo. Se você ainda não

executou a aplicação, sua pasta `android` não existe, mas, executando `ionic capacitor build android`, ela será criada sem a execução da aplicação, porém abrirá o Android Studio.

```
<!-- abaixo de <application> -->
<meta-data android:name="com.google.android.geo.API_KEY"
android:value="AIzaSyBLvLPmcOqHDMsE1k_gc5SEAQdnvapeKOX" />

<!-- abaixo de </application> -->
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"
/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-feature android:name="android.hardware.location.gps" />
```

Também precisamos realizar um registro no arquivo `\android\app\strc\main\java\io\ionic\starter\MainActivity.java`. Veja a implementação completa na sequência.

```
package io.ionic.starter;

import com.hemangkumar.capacitorgooglemaps.CapacitorGoogleMaps;
import com.getcapacitor.BridgeActivity;
import android.os.Bundle;


public class MainActivity extends BridgeActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      registerPlugin(CapacitorGoogleMaps.class);
    }
}
```

Para finalizarmos as configurações, precisamos implementar o código a seguir nos arquivos indicados.

```
// src/enviroments/environment.prod.ts e src/enviroments/environment.ts

export const environment = {
  production: true,
```

```
  mapsKey: 'AIzaSyBLvLPmcOqHDMsE1k_gc5SEAQdnvapeKOC'
};
```

Na figura a seguir, veja a aplicação em execução no emulador Android.

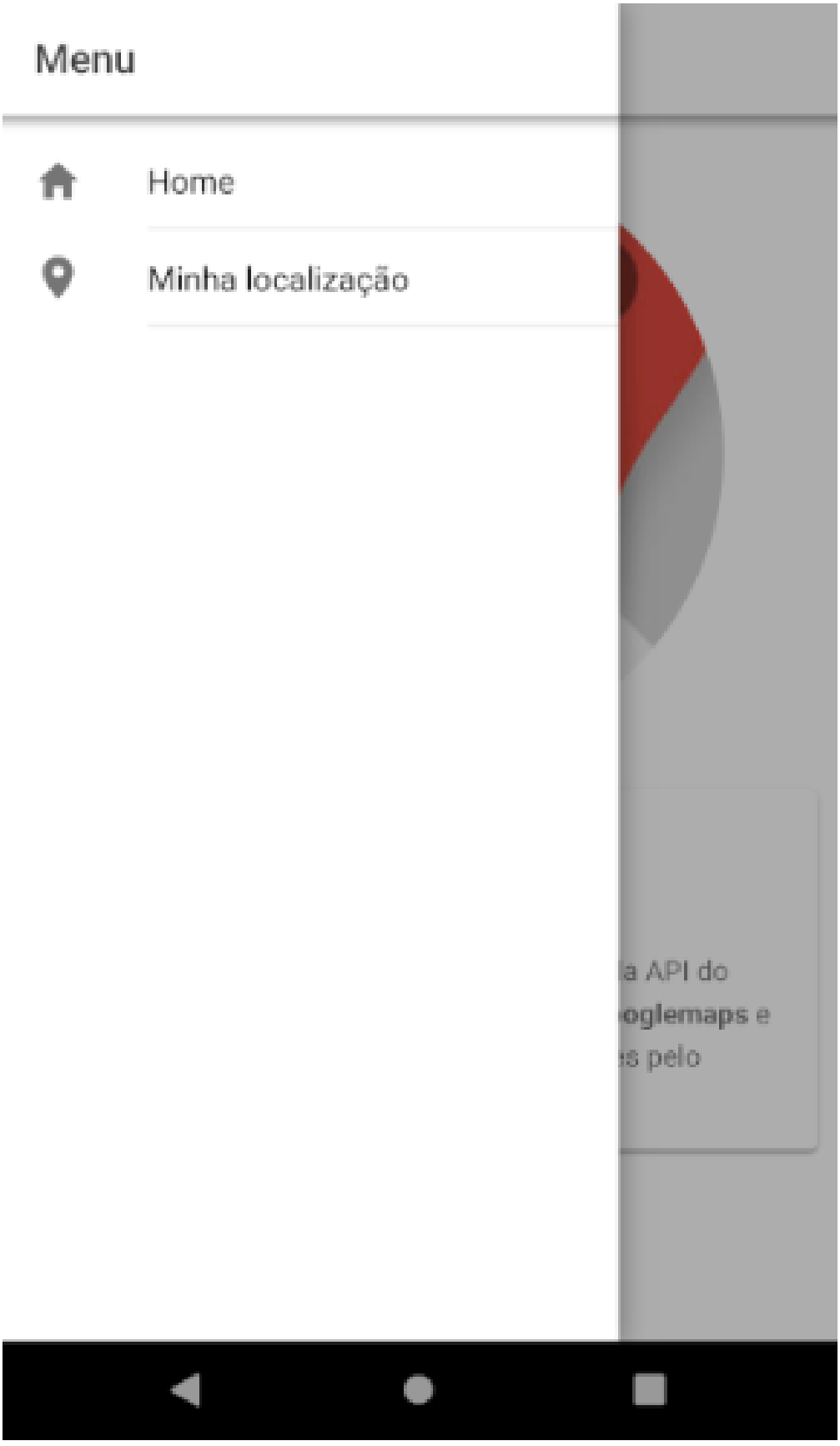

Menu
Home
Minha localização
a API do
oglemaps e
s pelo

Figura 10.6: Aplicação com menu em execução no Android

Para ajudar, trago na sequência as instruções a serem executadas no terminal.

```
// Preparação do projeto
npm run build

// Execução
ionic capacitor run android -l --external
```

10.4 Inserção do mapa na aplicação

Nosso primeiro passo é criar uma página que será renderizada quando o usuário optar por `Minha localização` no menu de opções da aplicação. Sendo assim, no terminal, digite a instrução a seguir.

```
ionic generate page mylocation
```

Essa nossa página exibirá o mapa do Google Maps por meio de nosso plugin, que instalamos no início do capítulo. Vamos então às implementações necessárias para os artefatos, no mesmo molde anterior, para `home`.

mylocation.page.html

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start" color="primary">
      <ion-back-button defaultHref="/home"></ion-back-button>
    </ion-buttons>
    <ion-title>
      Minha localização
    </ion-title>
  </ion-toolbar>
</ion-header>
```

```
<ion-content padding>
  <div id="map" #map>
  </div>
</ion-content>
```

Temos a configuração básica de uma página com a exibição do menu de opções e, na área de conteúdo, um `<div>`, que terá seu conteúdo renderizado por meio de nossa classe de estilo para este template.

mylocation.page.scss

Precisamos criar uma classe CSS para nosso div `map`, pois um mapa precisa de pelo menos uma área 100px x 100px para sua renderização. E também vamos aproveitar para outras configurações.

```
#map {
    width: 100%;
    height: 100%;
    border: 3px solid red;
}
```

mylocation.page.ts

Por termos um relativo código a implementar para este arquivo, vamos ver esta implementação por partes, sendo a primeira a que se segue.

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { CapacitorGoogleMaps } from '@capacitor-community/capacitor-
googlemaps-native';
import { Geolocation } from '@capacitor/geolocation';


@Component({
  selector: 'app-mylocation',
  templateUrl: './mylocation.page.html',
  styleUrls: ['./mylocation.page.scss'],
```

```
})
export class MylocationPage{

  @ViewChild('map') mapView: ElementRef;

  ionViewDidEnter() {
    this.createMap();
  }

  async createMap() {

  }

  ionViewDidLeave() {
    CapacitorGoogleMaps.close();
  }

  async showCurrentPosition() {

  }
}
```

Antes do construtor, temos a definição de um objeto, que será uma referência ao `<div>` que terá como conteúdo nosso mapa. Na sequência, quatro métodos: `ionViewDidEnter()`, que invocará o `createMap()`, o `ionViewDidLeave()`, que fechará o mapa que criaremos, e o `showCurrentPosition()`, que exibirá a posição atual do dispositivo no mapa. Vamos agora à implementação do `createMap()`. Veja na sequência.

```
async createMap() {
    const boundingRect =
this.mapView.nativeElement.getBoundingClientRect() as DOMRect;

    CapacitorGoogleMaps.create({
      width: Math.round(boundingRect.width),
      height: Math.round(boundingRect.height),
      x: Math.round(boundingRect.x),
      y: Math.round(boundingRect.y),
```

```
      latitude: -25.3009031,
      longitude: - 54.1143351,
      zoom: 16
    });

    CapacitorGoogleMaps.addListener('onMapReady', async () => {
      CapacitorGoogleMaps.setMapType({
        type: "normal"
      });
    });
}
```

A primeira instrução define uma área retangular para a exibição do mapa, com base na área recuperada para o `<div>` no dispositivo. Observe que atribuo um valor inicial para `latitude` e `longitude`. Na documentação, este procedimento não é necessário, porém, se não informarmos um valor para `latitude` e `longitude` no `create()`, a aplicação deixa de executar. Você pode aqui invocar o método `getCurrentPosition()`, que veremos a seguir para definir esta posição inicial.

O mapa pode levar um tempo para ser criado, o que nos leva a um `listener`, implementado na sequência ao `create()`. Quando o evento `onMapReady` ocorrer, `setMapType()` é invocado, definindo o tipo do mapa como `normal`. Ainda existem as opções `hybrid`, `satellite` e `terrain`. Com isso ocorrendo, o método `showCurrentPosition()` é invocado, renderizando o mapa exibindo a posição atual do dispositivo nele. Você pode, se quiser, aumentar ou diminuir o valor de `zoom`.

10.5 Identificação da localização do dispositivo

O mapa exibido pela página criada na seção anterior tinha a localização e exibição de um local específico, o câmpus da UTFPR

em Medianeira, local em que trabalho. Mas o nome do componente é `mylocation`. Deveríamos então pensar em exibir no mapa a nossa localização, ou seja, a localização de nossos dispositivos. Vamos fazer isso, com algumas mudanças em nossos artefatos. A solução não é a mais bonita, mas nosso objetivo é didático.

Nossa primeira alteração será visual, onde inseriremos um botão em nosso mapa que, ao ser pressionado, buscará por nossa localização geográfica e a representará em nosso mapa. Para isso, antes do `<div>` de nosso mapa, no template HTML, insira o código a seguir. Note que apenas inserimos um `<ion-button>` com algumas configurações e ligação com um método a ser executado quando for pressionado.

```
<div>
    <ion-button class="nopadding" shape="round" color="primary"
size="small" expand="block"
      (click)="showCurrentPosition()">
      Mostre minha localização
    </ion-button>
</div>
```

Agora vamos começar a implementar o necessário em nossa classe `MylocationPage` para que nossa localização possa ser exibida no mapa. A primeira parte está a seguir, logo abaixo da declaração de `mapView`.

```
loading: any;
```

Quando pressionarmos o botão, precisaremos mostrar para o usuário alguma coisa na tela do dispositivo, para ele não ter a impressão de congelamento da aplicação. Faremos uso do já conhecido `LoadingControler`, que precisamos injetar em nosso construtor, tal qual é mostrado na sequência.

```
constructor(private loadingCtrl: LoadingController,) { }
```

Será preciso incluir essa importação do `'@ionic/angular'` no início de nossa classe e, logo após isso, vamos implementar o método ligado

com `click` de nosso botão, na página.

Vamos agora implementar o método para recuperar a posição atual do dispositivo e atribuir uma marcação a este local? Veja a implementação para `showCurrentPosition()` trazida na sequência, veja o código. Observe a invocação ao `addMarker()`, responsável por inserir um marcador (pin) no mapa. Para que o mapa e sua localização sejam exibidos, precisamos invocar ainda o `setCamera()`. Veja que nele podemos também alterar o `zoom`.

```
async showCurrentPosition() {
    this.loading = await this.loadingCtrl.create({
      message: 'Por favor, aguarde...'
    });
    await this.loading.present();

    const coordinates = await Geolocation.getCurrentPosition();

    CapacitorGoogleMaps.addMarker({
      latitude: coordinates.coords.latitude,
      longitude: coordinates.coords.longitude,
      title: 'Achei você',
      snippet: 'Este é o local aproximado que você está, certo?'
    });

    this.loading.dismiss();
    CapacitorGoogleMaps.setCamera({
      latitude: coordinates.coords.latitude,
      longitude: coordinates.coords.longitude,
      zoom: 16,
      bearing: 0
    });
}
```

Para testar a aplicação no iOS, de acordo com a documentação, basta invocar a inicialização do plugin e eu optei por fazer isso logo no `app.component.ts`, como mostro na sequência.

```
constructor() {
    CapacitorGoogleMaps.initialize({
      key: environment.mapsKey
    });
}
```

A figura a seguir apresenta a aplicação em execução nas duas plataformas. Lembre-se de permitir acesso à localização em seu dispositivo e confirmar o uso quando o App solicitar.

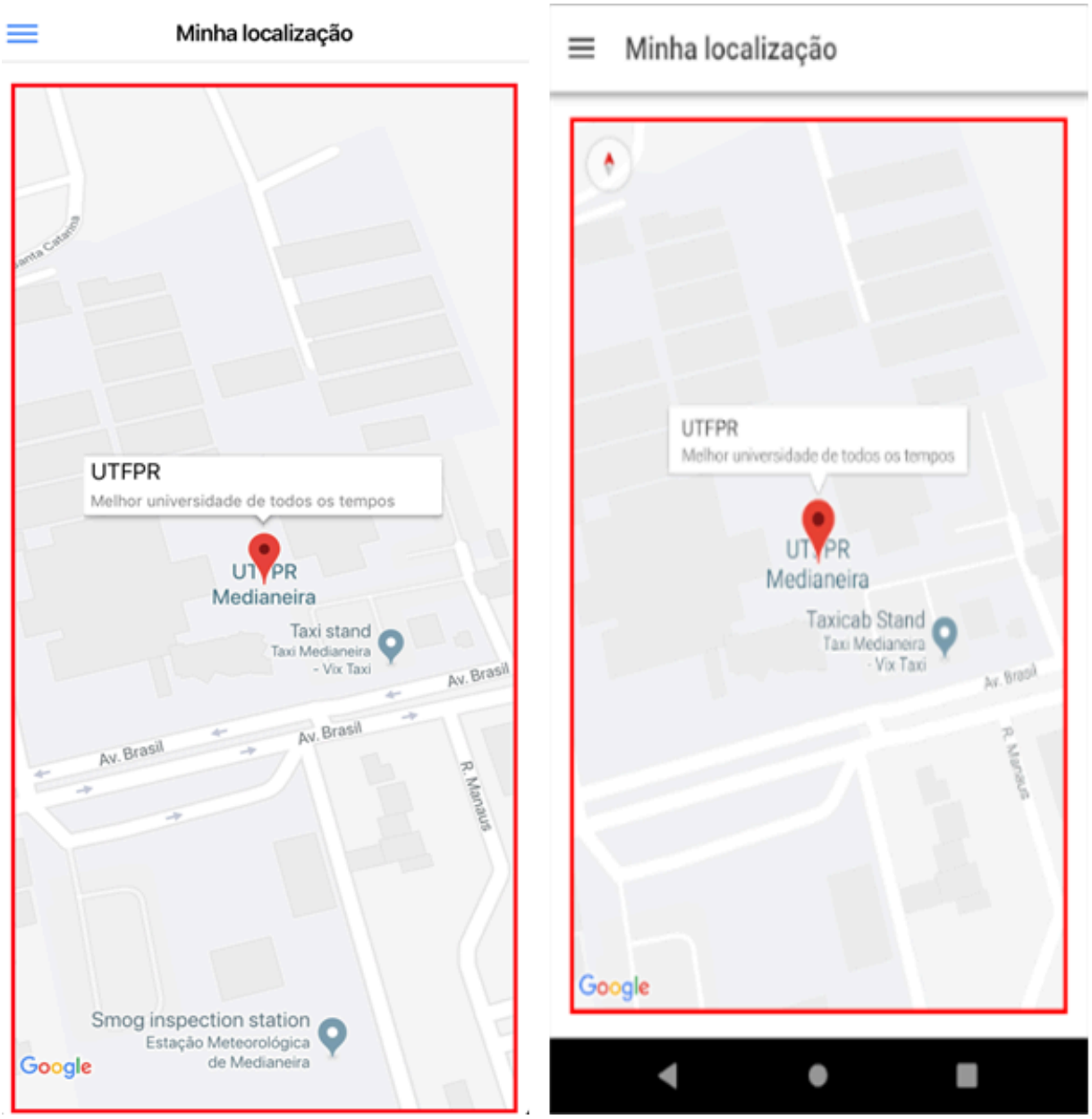


Figura 10.7: Mapa sendo exibido nos emuladores

10.6 A busca por um endereço

Na seção anterior, iniciamos um mapa com uma localização baseada em Latitude e Longitude, depois, buscamos a localização de nosso

dispositivo e vimos que essa nova localização também é exibida na mesma medida. Mas, normalmente, quando buscamos auxílio de um mapa, fazemos uso de um endereço para isso. É o mais usual. Vamos então criar uma nova página. Veja a instrução a seguir, que deve ser inserida em seu terminal.

```
ionic generate page lookforaddress
```

Essa instrução gerará todos os artefatos que precisamos e também a configuração de rotas, mas, em nosso `app.component.ts`, precisamos inserir essa nova opção ao usuário, em nosso objeto que representa as páginas. Vamos inserir lá o código a seguir.

```
,
{
  titulo: 'Localizar endereço',
  url: '/lookforaddress',
  icon: 'navigate'
}
```

Vamos adaptar os arquivos criados e começando com nosso componente HTML, o `lookforaddress.page.html`. Veja-o na sequência. Estamos utilizando o já conhecido `<ion-grid>`, pois precisaremos de um campo de entrada, `<ion-input>`, para o usuário digitar o endereço. Veja que estamos utilizando `#endereco` para que este componente esteja disponível em nossa classe.

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start" color="primary">
      <ion-back-button defaultHref="/home"></ion-back-button>
    </ion-buttons>
    <ion-title>
      Localização por endereço
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content padding>
```

```
  <div id="map" #map>
    <ion-grid>
      <ion-row>
        <ion-col col-9>
          <ion-input #address id="address" type="text"
placeholder="Endereço"></ion-input>
        </ion-col>
        <ion-col col-3>
          <ion-button class="nopadding" shape="round" color="primary"
size="small" expand="block"
            (click)="findAddress()">
            Localizar
          </ion-button>
        </ion-col>
      </ion-row>
    </ion-grid>
  </div>
</ion-content>
```

Precisamos definir características mínimas para nosso `map`. Em nosso arquivo de estilos, o `scss`, insira as seguintes instruções.

```
#map {
    width: 100%;
    height: 100%;
    border: 3px solid red;
}
```

Vamos à implementação da classe `LookforaddressPage`, o arquivo `lookforaddress.page.ts`, que será o responsável pela renderização e controle da interação com o usuário. Vejamos os códigos aos poucos.

```
// antes do construtor
loading: any;
@ViewChild('address') address: IonInput;
@ViewChild('map') mapView: ElementRef;
```

Conhecemos esse código da implementação anterior e podemos lembrar do `endereco` de outras implementações, onde fazemos a

ligação de um componente visual com nossa classe.

Temos também nosso construtor, que é semelhante ao que implementamos no exemplo anterior, mas, para facilitar, trago-o todo na sequência.

```
constructor(
    public loadingCtrl: LoadingController,
    public toastCtrl: ToastController
) { }
```

Com a estrutura básica implementada, precisamos focar agora na busca do endereço e exibição do mapa com a localização do endereço informada, localizada e identificada no mapa. Ocorre que em minhas buscas não identifiquei um plugin Capacitor para esta funcionalidade, mas existem plugins Cordova e pacotes nativos. Eu não quis transformar nossa aplicação Capacitor em Cordova, pois o recomendado é Capacitor. Assim, optei em utilizar os pacotes nativos, que são um tipo de ponte para os plugins Cordova. Execute as instruções a seguir em seu terminal.

```
npm install cordova-plugin-nativegeocoder
npm install @awesome-cordova-plugins/native-geocoder
```

Com os pacotes necessários instalados, vamos implementar o método responsável pela criação do mapa. Note que, diferente do que fizemos anteriormente, o método agora recebe a latitude e longitude que deverá ser utilizada na criação dele. Veja também que já implementamos o fechamento do mapa para quando a visão for fechada, imports e a injeção do pacote no construtor.

```
import { NativeGeocoder, NativeGeocoderResult, NativeGeocoderOptions }
from '@awesome-cordova-plugins/native-geocoder/ngx';

private nativeGeocoder: NativeGeocoder,

async createMap(latitude: number, longitude: number) {
    const boundingRect =
this.mapView.nativeElement.getBoundingClientRect() as DOMRect;
```

```
    CapacitorGoogleMaps.create({
      width: Math.round(boundingRect.width),
      height: Math.round(boundingRect.height),
      x: Math.round(boundingRect.x),
      y: Math.round(boundingRect.y),
      latitude: latitude,
      longitude: longitude,
      zoom: 16
    });

    CapacitorGoogleMaps.addListener('onMapReady', async () => {
      CapacitorGoogleMaps.setMapType({
        type: "normal"
      });
    });
}

ionViewDidLeave() {
    CapacitorGoogleMaps.close();
}
```

Como trabalharemos agora a invocação para este método? Isso mesmo, vamos implementar o método que está ligado ao botão ao lado do campo que receberá o endereço. Veja esta nova implementação na sequência. Observe que usamos o `LoadingController` para que o usuário veja que há um processamento em execução, configuramos as opções para o retorno à invocação do `forwardGeocode()`, que recebe o endereço e essas opções. Em caso de sucesso ou falha, o `LoadingController` é oculto e, em caso de falha, o `ToastController` é invocado. Já veremos o método `showToast()`.

```
async findAddress() {
    this.loading = await this.loadingCtrl.create({
      message: 'Por favor, aguarde...'
    });
    await this.loading.present();

    let options: NativeGeocoderOptions = {
```

```
      useLocale: true,
      maxResults: 5
    };

    this.nativeGeocoder.forwardGeocode(this.address.value.toString(),
options)
      .then(async (result: NativeGeocoderResult[]) => {
        this.loading.dismiss();
        await this.createMap(Number(result[0].latitude),
Number(result[0].longitude));
      })
      .catch(async (error: any) => {
        this.showToast('Não encontrado ' + error);
        await this.loading.present();
     });
}
```

Vamos aqui então apresentar o método `showToast()` utilizado no código anterior. Veja-o na sequência.

```
async showToast(message: string) {
    const toast = await this.toastCtrl.create({
      message,
      duration: 4000,
      position: 'middle'
    });

    toast.present();
}
```

Também se faz necessário, no `app.module.ts` injetarmos no `providers` o `NativeGeocoder` , com o mesmo `import` visto anteriormente.

Execute a aplicação e escolha no menu a nova opção. Não trarei figuras aqui, por serem semelhantes às que já vimos. A mudança está na inserção de uma caixa de texto, para que o usuário possa informar o endereço. O padrão de digitação é o mesmo para o Google Maps. Caso você receba algum erro (`grpcn failed`), referente à execução da busca, tente novamente. Há uma `issue` referente a

isso, mas pelo que é identificado, está ligado o dispositivo e o o GPS dele.

Bem, você pode verificar que o mapa é exibido com o endereço digitado ao centro dele. O que acha de colocarmos uma marca neste endereço, para ficar mais compreensível que foi encontrado? Veja o código a seguir, que deve ser inserido após a invocação ao `createMap()`.

```
CapacitorGoogleMaps.addMarker({
    latitude: latitude,
    longitude: longitude,
    title: 'Aqui está',
    snippet: 'Está certa a localização?'
});

CapacitorGoogleMaps.setCamera({
    latitude: latitude,
    longitude: longitude,
    zoom: 16,
    bearing: 0
});
```

Muitos outros serviços são possíveis quando se trabalha com mapas, mas quis aqui apenas aguçar sua curiosidade para isso. Verifique a documentação dos plugins e pacotes que utilizamos e poderá ver diversas outras funcionalidades lá.

10.7 Ícone da aplicação e SplashScreen

Você viu, durante todo o livro, que o ícone da aplicação distribuída e a figura da tela de inicialização da aplicação (SplashScreen) é a padrão do ícone. Chegou a hora de mudarmos isso. Precisaremos instalar um novo pacote. Veja a instrução na sequência.

```
npm install -g cordova-res
```

Agora, na raiz de nosso projeto, vamos criar uma pasta chamada `resources`. Dentro dessa pasta, uma nova, chamada `android`. Dentro da pasta `resources`, você colocará um arquivo chamado `icon.png`, de dimensão 1024x1024 e um arquivo chamado `splash.png`, de dimensão de 2732x2732. Dentro da pasta `resources\android`, mais dois arquivos: `icon-background.png`, que representa a cor de fundo do ícone, eu usei a cor branca; e outro arquivo, chamado `icon-foreground.png`, que terá a imagem que representa seu ícone. Ambos arquivos com dimensão 432x432. Eu deixei o último arquivo com fundo transparente.

Com as imagens criadas e armazenadas conforme as orientações, no terminal, execute as instruções a seguir e após isso execute sua aplicação. Veja a imagem de splash sendo exibida e veja o ícone que representa o aplicativo.

```
cordova-res ios --skip-config --copy
cordova-res android --skip-config --copy
```

Conclusão

Terminamos o livro. Neste nosso último contato de aprendizado com o Ionic vimos algumas possibilidades de trabalho com mapas. Visualizamos um mapa básico, buscamos a localização de nosso dispositivo e ainda procuramos por um determinado endereço. Não chegamos nem perto de esgotar o assunto. Existe muita possibilidade e recursos nesta área.

Fechando o capítulo, trouxemos a possibilidade de alterar nosso ícone da aplicação quando instalada em dispositivos e emuladores, assim como trocar a figura apresentada quando a aplicação é executada, a conhecida SplashScreen.

Tivemos uma boa experiência no estudo deste framework. Foi possível aprendermos muito juntos. Siga em seus estudos. Procure aprofundar seus conhecimentos nos apresentados no livro, descubra

novos recursos e componentes que não pudemos ver aqui. E lembre-se de se divertir, sempre.

Capítulo 11
Os estudos não param por aqui

Os dispositivos móveis já fazem parte do dia a dia da maioria da população. Você, como programador(a) ou desenvolvedor(a) não pode perder essa onda.

São duas as maiores plataformas móveis atualmente disponíveis (iOS e Android) e várias são as ferramentas para desenvolvimento de aplicações para elas, quer seja de forma nativa ou híbrida. Neste livro, você teve acesso à metodologia de implementação de aplicações que podem ser executadas nas plataformas descritas, por meio do Ionic, utilizando HTML e TypeScript.

Fizemos um passeio sobre grande parte dos controles disponibilizados pelo Ionic. Também testamos as aplicações em emuladores e em dispositivos físicos, o que nos possibilitou uso de recursos como câmera e álbum de fotografia.

Para a execução nos emuladores e dispositivos, fizemos uso do CAPACITOR. Framework que migra nossas aplicações Ionic para as plataformas específicas e desejadas.

Como o foco da aplicação desenvolvida no livro foi comercial, não podíamos deixar de trabalhar com acesso a base de dados, o que fizemos por meio do SQLite e Firestore. Uma característica comum em aplicações móveis é o sincronismo com aplicações na nuvem, e vimos que o Firestore trata isso com maestria.

O livro não esgotou os temas trabalhados. É preciso dedicação para investigação e descoberta de novas tecnologias e recursos. Tenho certeza de que ele foi mais do que um pontapé inicial para o seu desenvolvimento, no que se diz respeito a aplicações móveis.

Espero que, com o conteúdo que trabalhei, tenha sido despertada em você uma grande curiosidade e que o livro tenha desmitificado o

desenvolvimento de aplicativos para dispositivos móveis. Agora é bola para a frente na evolução.

Obrigado pela companhia e sucesso.